University of Zurich[UZH]

Department of Informatics

Combining Linear and Relational Algebra for Analytical Query Processing with Contextual Information

Dissertation submitted to the Faculty of Business,
Economics and Informatics
of the University of Zurich

to obtain the degree of
Doktorin der Wissenschaften, Dr. sc.
(corresponds to Doctor of Science, PhD)

presented by

Oksana Dolmatova

from Saint Petersburg, Russia

approved in February 2021

at the request of

Prof. Dr. Michael H. Böhlen
Prof. Dr. Stefan Manegold

University of Zurich[UZH]

Department of Informatics

The Faculty of Business, Economics and Informatics of the University of Zurich hereby authorizes the printing of this dissertation, without indicating an opinion of the views expressed in the work.

Zurich, February 17, 2021

The Chairman of the Doctoral Board: Prof. Dr. Thomas Fritz

Abstract

Data that is stored in relational databases often includes a numerical part that should be analyzed. Typical examples of such data are time series, stock exchange information, user ratings, or sales data. A salient property of such data is that, along with numerical values that must be analyzed, it includes additional contextual information. This information is an important part of the data, as it gives meaning to the numerical values. Contextual information includes a relation schema and values of some attributes, which specify units, give names, and add context to the numbers. The growing demand to analytically process data that include a combination of numerical values along with their context poses a challenge for existing relational databases and state-of-the-art solutions for data analysis. Typical analytical queries include a mixture of linear and relational algebra operations. Additionally, some queries do not have closed-form solutions and involve iterative computations, such as gradient descent computations. Thus, the main challenge is to support a combination of linear and relational operations over data with contextual information.

Linear and relational algebras operate on matrices and relations, respectively. We address the issue of mixed queries, which include operations from both algebras, by closing a logical gap between relations and matrices. Our solution identifies parts of a relation with different semantics: Contextual information and an application part. These parts are equally important, but have different purposes and must be treated differently during data analysis.

We show the importance of contextual information and suggest two novel concepts, the relational matrix algebra and shape preserving iterations. These concepts allow to perform data analysis

over relations while preserving contextual information in the result, i.e., each result relation includes all information that is required to make the result interpretable. The relational matrix algebra extends the standard relational algebra with matrix operations defined over relations. Shape preserving iterations allow to apply iterative methods to relations. The relational matrix algebra combined with shape preserving iterations allows to perform modern data analysis over relations.

Our approach is integrated into the kernel of the column-oriented database MonetDB to show the practical feasibility and demonstrate the effectiveness of our ideas. We explain the challenges that arose during the implementation and the architectural decisions we made to overcome these challenges. We leverage MonetDB's internal structures and its query processing pipeline to produce a system that can outperform existing solutions. Our work includes an extensive evaluation of our integration and a comparison with state-of-the-art techniques.

At the end, we design, implement, and evaluate a database solution that preserves the advantages of the relational model, such as relational optimizations, and allows to statistically analyze data stored in relations.

To my family

Acknowledgments

First and foremost, I would like to express my gratitude to my advisor Prof. Dr. Michael Böhlen for the opportunity to pursue the PhD in the Database Technology Group at the University of Zürich. I am very thankful for his support, constructive discussions during the meetings, and the effort he invested in our work. I admire his attention to details and ability of getting to the essence of things.

A special thanks to Prof. Nikolaus Augsten for the substantial effort and time he put in our papers.

I would like to thank Prof. Dr. Stefan Manegold for agreeing to be a co-advisor of my thesis and Prof. Dr. Thomas Fritz, for chairing the thesis defense.

I thank the MonetDB team for their support and invaluable expertise.

I want to thank my former advisor Prof. B. Novikov and teachers D. Maximov and S. Rukshin for showing me the beauty of computer science and mathematics.

Many thanks to all my current and former colleagues from the Database Technology Group at the University of Zürich for the discussions, feedback, and help.

I would like to thank my family for their unconditional love and support.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Background

Database management systems (DBMSs) are the main tool to store, organize, and query large sets of data, and the amount of data that is being produced and must be stored in a DBMS is growing fast. Along with the size of the data, the demand to analyze the data with techniques that are based on mathematical and statistical algorithms is increasing quickly.

Many data analysis tasks over structured data require a mixture of relational algebra operations, such as joins and selections, and linear algebra operations, such as inversion, multiplication, and various decompositions. The expression $r \bowtie (s * s^t)$, which applies a join ($\bowtie$), matrix multiplication ($*$), and matrix transpose (^{t}) to relations r and s, is a typical example of mixing the two algebras. In addition, some analytical tasks do not have a solution that can be formulated as a closed formula, and their solutions are based on iterative methods. Iterative methods generate a sequence of approximations computed from an initial guess until the desired quality threshold is reached.

Although databases store and organize data, the classes of queries supported by relational database systems are usually limited to data selection, combining, and aggregation. This limitation prevents database systems from becoming a platform for modern data processing, because the standard relational algebra is not enough to perform mixed queries and to process data as required by data scientists.

As a result, additional tools and environments are involved in data processing. Often data scientists are forced to export data from a database and use third party tools to run analytical queries, as well as to import results back in order to store, manage, and query their findings. Moving data is not efficient and requires additional efforts and time.

There are many attempts to address the problem of combining the relational and the matrix models. Some of them focus on integrating new objects (e.g., an array) into the relational model. This approach enables analytical computations within a database system, but the gap between a relation and a matrix remains, as different sets of operations are defined over these two objects. To combine the two algebras in one query, one must translate a relation to a matrix and back. Some solutions emulate relations via arrays or vice versa. Other approaches offer a combination of existing systems that is transparent to the end user. While transparent, it addresses the problem only on the implementation level, without solving the logical mismatch between a relation and a matrix. We are the first who fundamentally address the problem of how to bridge the logical gap between relations and matrices.

The main goal of this thesis is to combine the relational and the linear algebras into a single model. This model has only one object type, which is a relation with contextual information, and puts operations from both algebras on the same level. In other words, the model supports expressions that include operations from the relational and linear algebras, and these operations are defined over relations. Such expressions allow to use operations of one algebra applied to the result of another algebra operation in a straightforward manner.

We believe that the problem of combining the two algebras can be solved in an insightful, precise, and intuitive way. We offer an extension of the relational model with linear algebra operations and iterations and its implementation in a database. We show that the extended relational model is well-suited for modern data analysis.

1.2 Challenges

In this thesis we address two challenges: Combining the relational and linear algebras, and deeply integrating iterations into the relational model.

The first challenge comes from the differences between the objects over which the two algebras are defined. The linear algebra and iterative methods work with matrices. In databases and in the relational algebra a relation is the main object. Both a relation and a matrix[1] in their natural form are two-dimensional structures that can be represented as tables. At the same time, they also have important differences. First, a relation is an unordered set of tuples, while a matrix has a precise and fixed order of rows. Second, matrices, over which standard linear algebra operations are defined, include only numerical values, but relations are heterogeneous and their attributes may have values from different domains.

Example 1. Consider matrix *m* and relation *r* given in Figure 1.1. Graphically both objects are tables, and they include the same numerical values.

m

	1	2	3
1	5	53	6,100
2	7	70	11,900
3	9	40	3,100
4	2	120	12,200

r

Date	Item	Rebate	Number	Profit
2019-03-21	Moscow Bread	5	53	6,100
2019-03-21	Raisin Cake	7	70	11,900
2019-03-22	Salt Sticks	9	40	3,100
2019-03-22	Moscow Bread	2	120	12,200

Figure 1.1: Matrix versus relation

Gray cells represent matrix indices that identify and fix the order of rows and columns in matrix *m*. Relation *r*, on the other hand, is unordered, and its tuples can be swapped without changing semantics. Relation *r* and its schema include additional information (e.g., attribute names Date and Profit, and attribute values 'Salt Sticks' and '2019-03-21'). We call these values contextual information.

Row and column indices are not part of *m* and include only order information. For example, from the indices we can infer that number 70 belongs to the second row of *m*. Similar to matrix indices, contextual information might be used to impose an order on the tuples. For example, number 70 also belongs to the second tuple of *r* ordered by Date and Item. However, in addition

[1] We do not consider tensor algebra in this thesis

to ordering, values in contextual information help to interpret numerical values in *r*. Although the numerical information included in *r* and *m* is the same, only *r* provides enough context to make numerical information meaningful. □

The second challenge comes from the limitation of the declarative paradigm of SQL. SQL describes the result with a closed formula expression and does not specify the steps to compute the result. This is the key property of the existing relational optimizations. However, this approach is not well-aligned with iterative methods that are used when a closed formula is very expensive to compute or is not available, such as in gradient descent computations.

The closest concept to iterations in SQL are recursive queries. A recursive query can be exploited to perform iterative methods over relations. However, recursive queries are not suitable for iterative methods due to their properties. In iterative methods, an approximate result is iterated over and over until a certain threshold of quality is reached. The shape of the approximate result stays fixed through all steps of the computation. In recursive queries the exit condition is additive in nature (e.g., the query stops when the closure of an input set has been computed), and all intermediate results are part of the final result relation. Thus, using recursive queries for iterative methods is not only inefficient performance-wise, but it also makes the translation from an iterative method to an SQL query challenging.

1.3 Contributions

We offer a model that supports the combination of relational queries and modern data analysis over relations and that preserves contextual information throughout query processing.

This thesis makes the following contributions to the field of analytical databases:

1. It identifies two important parts of a relation: Contextual information and an application part. The application part includes numerical information, and the contextual information gives meaning to the values in the application part. It demonstrates that contextual information is significant and must be preserved throughout analytical processing.

2. It introduces the relational matrix algebra (RMA). RMA is an extension of the standard relational algebra with matrix operations over relations (relational matrix operations). The relational matrix algebra is closed, i.e., it takes relations as input and returns relations.

RMA allows to easily combine standard relational and relational matrix operations in one expression. The thesis proposes a systematic way to preserve sufficient contextual information in result relations via origins. Origins are constructed from the contextual information of the input relations and uniquely identify and describe each cell in the result relation. We prove that the relational matrix algebra operations yield relations with origins.

3. It introduces shape preserving iterations over relations. The iterations include an iteration body, an exit condition, and an iterated relation. Values of the iterated relation are refined in each step of shape preserving iterations. Shape preserving iterations extend the set of tasks that can be computed within the relational model with iterative methods. To support algorithms that iteratively refine a matrix with initially random values, such as gradient descent, we introduce random initialization of iterated relations. This is a general approach that creates relations with random numerical values and proper contextual information. The random initialization is based purely on RMA expressions. Shape preserving iterations applied to randomly initialized relations deliver result relations with origins.

4. It identifies necessary extensions and components required to support relational matrix algebra operations in a column-oriented database. The thesis provides an implementation of selected matrix operations in the kernel of MonetDB and illustrates this integration with examples. The integration leverages internal MonetDB data structures for the efficient processing of the contextual information and the application part. It explains how to extend the internal MonetDB query tree, which is based on column operations, with shape preserving iterations. The integration supports relational optimizations in an iteration body and an exit condition of shape preserving iterations.

In this thesis we address the above mentioned challenges starting with an application scenario that illustrates a problem, which is followed by an analysis and a precise definition of the problem. We propose an approach and then study it and its properties, implement it to demonstrate its feasibility, and offer an extensive evaluation to confirm the effectiveness of the approach and the theoretical results. The main parts of the solutions proposed in this thesis are implemented as a part of the column-oriented database MonetDB and made available as open source software[2].

The rest of this section elaborates the contributions of this thesis in detail.

[2]https://github.com/oksdolm/RMA/blob/master/README.md

1.3.1 Contextual Information

The first contribution of this thesis is the approach that identifies parts of a relation that have different semantics. We offer a logical decomposition of a relation with numerical information into two parts: An application part and contextual information. The application part consists only of the values of the numerical attributes and is the information that must be analytically processed. The contextual information includes all attribute names as well as the values of the attributes that are not included in the application part.

The term "contextual information" is essential for all contributions of this thesis. First, the values in the contextual information impose an order on the values in the application part to make it suitable for further analysis, i.e., the contextual information establishes the order of tuples in an otherwise unordered relation. Second, the values in the contextual information give meaning to the values in the application part. Thus, in this thesis we only consider relations that include contextual information and we maintain contextual information throughout data processing.

Example 2. Figure 1.2 illustrates relation r that stores information about sold backed items. For example, r states that 53 Moscow Breads were sold on 21st of March, 2019 with a rebate of 5% and a total profit of 6,100 dollars. This relation consists of contextual information and an application part. The values in the colored cells compose the contextual information. The values in the white cells are the application part.

r

Date	Item	Rebate	Number	Profit
2019-03-21	Moscow Bread	5	53	6,100
2019-03-21	Raisin Cake	7	70	11,900
2019-03-22	Moscow Bread	2	120	12,200
2019-03-22	Salt Sticks	9	40	3,100

Figure 1.2: The contextual information and the application part of a relation

The contextual information consists of three parts. The cyan part denotes the names of the attributes used for ordering. The blue part describes the values in the application part and determines the order of the tuples. In this example the tuples are shown in the order imposed by values in attributes Date and Item in ascending order. The purple part provides additional description of the values in the application part. For example, from the contextual information of r we can infer that 70 is the number of Raisin Cakes sold on 2019-03-21. □

1.3.2 The Relational Matrix Algebra

The second contribution of the thesis introduces the relational matrix algebra and origins. We take basic matrix operations such as multiplication and inversion (MMU and INV) and define corresponding relational matrix operations over relations (mmu and inv). The set of all relational and relational matrix operations is called the relational matrix algebra (RMA). The relational matrix algebra is closed with respect to relations and allows to easily translate mathematical formulas to algebra expressions.

Example 3. Consider a typical analytical query that applies a relational join to the result of a linear regression computation. The query requires to perform operations from different algebras: (1) linear regression computation from linear algebra and (2) join computation from relational algebra. In linear algebra, the expression $(\texttt{CPD}(m,m))^{-1} * \texttt{CPD}(m,n)$ computes the ordinary least squares (OLS) coefficients required by linear regression, where CPD is crossproduct, $^{-1}$ is inversion, and $*$ is multiplication. In relational algebra, $r \bowtie s$ computes the natural join between two relations, where relation r corresponds to the result of the OLS computation.

Figure 1.3a illustrates the RMA expression that combines the OLS formula and the join operation over relations. Relation $u1$ corresponds to matrix m and relation $u2$ corresponds to vector n. For simplicity Figure 1.3b illustrates the same expression step by step. With the relational matrix algebra, the mapping from matrix expressions to relational matrix expressions is straightforward. For example, matrix multiplication $*$ is translated to relational matrix multiplication $\texttt{mmu}_{V;V}$ and matrix inversion $^{-1}$ is translated to relational matrix inversion $\texttt{inv}_V$. The subscript that follows the name of an operation specifies the attributes used for ordering.

$$\texttt{mmu}_{V;V}(\texttt{inv}_V(\texttt{cpd}^V_{D;D}(u1,u1)), \texttt{cpd}^V_{D;D}(u1,u2)) \bowtie s$$

(a) One expression

$$\begin{aligned} u3 &\leftarrow \texttt{cpd}^V_{D;D}(u1,u1) \\ u4 &\leftarrow \texttt{inv}_V(u3) \\ u5 &\leftarrow \texttt{cpd}^V_{D;D}(u1,u2) \\ r &\leftarrow \texttt{mmu}_{V;V}(u4,u5) \\ t &\leftarrow r \bowtie s \end{aligned}$$

(b) Step by step

Figure 1.3: RMA expression for ordinary least squares and join computation

RMA allows to use operations from both algebras in one expression. The result of an operation of one algebra can directly be used as input for an operation from the other algebra. □

While defining the relational matrix algebra, we address the following questions: What contextual information should be included in the result? What properties should the result relation have? To answer these questions we introduce the concept of row and column origins. In essence, origins are the part of contextual information that describes and identifies values in the result relation. To define origins we leverage the following property of matrix operations: The result matrix cardinalities are determined solely by the cardinalities of the inputs and the operation. Based on this property we propose a principled solution to handle contextual information in the relational matrix algebra. The inheritance of the contextual information in a relational matrix operation depends on the inheritance of cardinalities of the corresponding matrix operation.

Example 4. Consider relation $s1$ given in Figure 1.4. Relation $s1$ states the ratios of carbohydrates, fats, and proteins in various baked items. Relation $s1$ is transposed using the relational matrix transpose `tra` that yields relation $s2$, i.e., $s2 = \texttt{tra}^{C}_{Item}(s1)$. After that, relation $s2$ is transposed again yielding relation $s3$, i.e., $s3 = \texttt{tra}^{Item}_{C}(\texttt{tra}^{C}_{Item}(s1))$.

The origins of relations $s2$ and $s3$ are marked by ellipses. All values inside an ellipse form together the origin for the relation. For example, the column origin of relation $s2$, i.e., (Bread, Cake), is composed of values in the order part of relation $s1$. The column origin describes columns, the row origin describes rows.

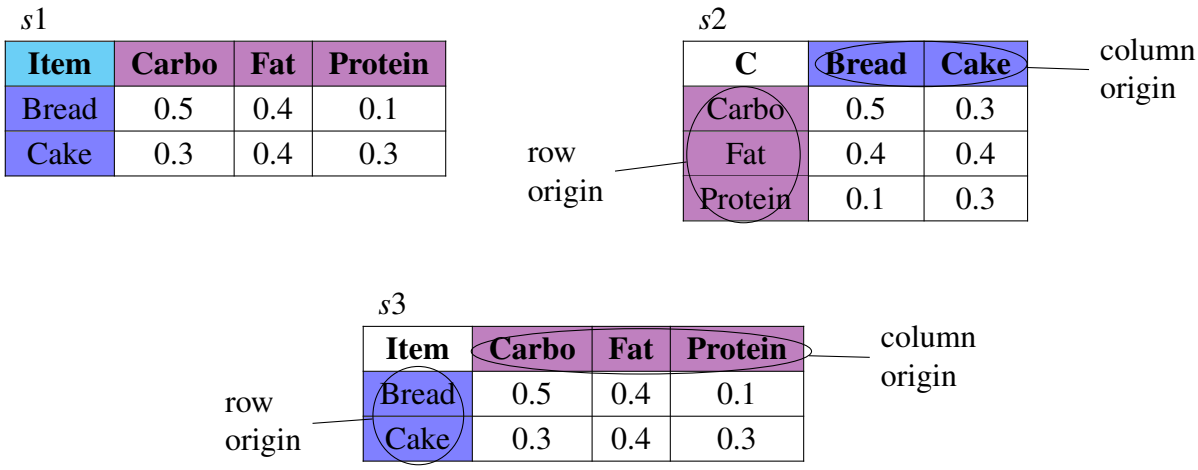


Figure 1.4: Origins in `tra` computation

The key observation in this example is that result relation $s3$ is equal to the fist input relation $s1$. This equality holds due to the presence of origins, i.e., proper contextual information, in each result relation. Notice that the origins of $s2$ are inherited from the contextual information of $s1$,

and the origins of $s3$ are inherited from $s2$. To summarize, we apply the relational transpose two times and get the initial relation. This property matches the property of the standard matrix transpose operation.

Since values in the contextual information are inherited from the input relations, relations $s1$, $s2$, and $s3$ are linked to each other. Additionally, $s2$ and $s3$ are interpretable standard relations that can also be used as input relations in other relational operations. □

The main result of the second contribution is that the relational matrix algebra operations yield result relations with origins.

1.3.3 Shape Preserving Iterations

The third contribution of this thesis adds shape preserving iterations to the relational model and SQL. Shape preserving iterations allow to use iterative methods over relations.

A shape preserving iteration takes input relations, returns a result relation, and consists of an iteration body and an exit condition. Each step of the iteration updates the values of one of the input relations. This input relation is the iterated relation and is returned as the result. The exit condition is evaluated after each computation of the iteration body. The shape preserving iteration stops when the exit condition is satisfied. Each shape preserving iteration has the following form: $I_r(Q^r, E)$, where r denotes the iterated relation, Q^r is the iteration body, and E is the exit condition.

Shape preserving iterations are defined through stable queries and relational predicates. A stable query is a relational matrix algebra expression that in the result relation preserves the number of attributes, the number of tuples, and key attributes of at least one input relations. A relational predicate is computed over the iterated relation. In shape preserving iterations $I_r(Q^r, E)$ there is the stable query Q^r, and the relational predicate E.

Shape preserving iterations have the following key properties. First, shape preserving iterations are in-place iterations. After each execution of the iteration body, a shape preserving iteration replaces the values in the iterated relation with the values computed with the iteration body. This is possible because the iteration body is a stable query, and the size of its input and result relations is fixed throughout the computation. Second, the iteration body is a relational expression that describes the result relation, and the exit condition is a relational predicate (also a relational

expression) that quantifies the quality of the result. This property enables existing relational optimizations for iterations.

Example 5. Consider relation t illustrated in Figure 1.5 and the example iteration $I_t(\pi_{A,B*0.1}(t), \vartheta_{\text{SUM}(B)}(t) < 0.1)$. This iteration is a shape preserving iteration that multiplies values in attribute B of iterated relation t by 0.1 until their sum is smaller than 0.1. The iteration body is a stable query (Q^t) since it preserves the shape of relation t and the values in the key attribute A.

Figure 1.5 illustrates the input iterated relation t, the iterated relation after the first step of the shape preserving iteration, i.e., $Q^t(t)$, and the result iterated relation t'. After executing the iteration body the exit condition is evaluated. For example, $\vartheta_{\text{SUM}(B)}(Q^t(t))$ is equal to 2, and, thus, the predicate evaluates to false. The iteration body and the exit condition are performed again and again. Finally, after the third step the predicate over t' evaluates to true, and the iteration stops.

t

A	B
Joe	5
Lee	6
Tom	9

$Q^t(t)$

A	B
Joe	0.5
Lee	0.6
Tom	0.9

...

t'

A	B
Joe	0.005
Lee	0.006
Tom	0.009

Figure 1.5: Example of a shape preserving iteration

The contextual information of relation t is preserved in relation $Q^t(t)$. Both relations have the same schema and the common attribute A. The contextual information remains unchanged throughout the computation and describes the values in attribute B in the result relation $t' = Q^t(Q^t(Q^t(t)))$. □

Typically, the iterated matrices in iterative methods are initially populated with randomly generated numbers. For example, to compute logistic regression with gradient descent, the initial coefficients matrix is randomly generated. Each step of the gradient descent iteration refines the values in this matrix. To address the problem of creating relations with a random application part and proper contextual information, we propose a random initialization of relations. The application part of a randomly initialized relation is generated according to the properties given by a user and corresponds to the iterated matrix. Its contextual information describes the final values in the application part and is inherited from the existing relations. Random initialization provides

contextual information for iterated relations. We prove that the set of relational matrix algebra operations is sufficient to create randomly initialized relations with proper contextual information. Thus, we leverage the introduced relational matrix algebra operations to create relations for shape preserving iterations.

Finally, we prove that shape preserving iterations applied to randomly initialized relations created with RMA expressions return result relations with origins.

1.3.4 Building a System

The last contribution of the thesis is a comprehensive system that implements our solutions. The starting point of our implementation is the column-oriented database system MonetDB. We describe how MonetDB's internal data structures and its query pipeline are leveraged to efficiently integrate relational matrix operations and iterations into the kernel of MonetDB. We integrate relational matrix algebra operations and shape preserving iterations into MonetDB, preserving the advantages of its engine, such as data compression, fast relational query processing, and available relational optimizations. We explain the challenges and architectural decisions that we made during the implementation.

The integration of relational matrix operations consists of two parts: Processing of contextual information and calculation of the matrix result. Contextual information processing is always performed internally in MonetDB. The column-oriented nature of MonetDB supports our idea of decomposing a relation into contextual information and an application part. Decomposing the input relations and composing the result relation are very efficient operations that work at the schema level and do not access the stored data. We offer two approaches to calculate the matrix result of a relational matrix operation. The first approach is a full integration of matrix algorithms inside the database kernel. It involves expressing a matrix algorithm as a sequence of column operations offered by MonetDB, which are also used for standard relational operations. The second approach delegates the computation of the matrix result to the Intel Math Kernel Library, which includes optimized math routines for scientific tasks.

We evaluate our integration by comparing the implementation with the state-of-the-art approaches. Our system outperforms existing solutions for the queries that require a mixture of operations from linear and relational algebra. We also study differences between the two integration methods of the matrix result computation. We explain the best use cases for the full

integration and for the delegation. The decision of which integration should be used depends on the amount of available main memory and on the complexity of the matrix operation, i.e., whether this operation can be efficiently expressed as a sequence of column operations.

The integration of shape preserving iterations into the kernel of MonetDB requires to extend the standard internal MonetDB query tree structure with an additional update operation and a loop structure node. The new update operation is used to update the values in the application part of the iterated relation, and the loop structure is needed to express the repetitions of the iteration body and the exit condition. To extend the query tree structure with shape preserving iterations we leverage the existing update node and the existing low-level control structures of MonetDB.

We show that shape preserving iterations can also be used in subqueries: The extended query tree can be part of another query tree. The standard and extended query trees are compatible and can be combined.

The system we built confirms our hypothesis that databases can serve as platforms for different types of operations over structured data. Our solution supports efficient data storage as well as modern data analysis.

1.4 Organization of the Thesis

This thesis is based on a collection of papers. Each chapter is self-contained and can be read in isolation.

Chapter 2 Motivation

based on the short paper: *"Preserving Contextual Information in Relational Matrix Operations.", O. Dolmatova, N. Augsten, and M. Böhlen, Proceedings of the 36th International Conference on Data Engineering (ICDE), Dallas 2020, 4 pages*

and on the paper: *"Iterations and Propensity Score Matching in MonetDB.", M. Böhlen, O. Dolmatova, M. Krauthammer, A. Mariyagnanaseelan, J. Stahl and T. Surbeck, Advances in Databases and Information Systems, 24th East European Conference (ADBIS), Lyon 2020, 14 pages*

Chapter 3 Building a System with Relational and Matrix Operations

based on the paper: *"A Relational Matrix Algebra and its Implementation in a Column Store.", O. Dolmatova, N. Augsten, and M. Böhlen, International Conference on Management of Data (SIGMOD), Portland 2020, 14 pages*

Chapter 4 Building a System with Iterations

based on the paper: *"Shape Preserving Iterations in a Column-Store.", O. Dolmatova and M. Böhlen, [Ready for submission]*

The original papers have been adjusted to make the chapters consistent throughout the thesis. For example, the notation of relational matrix operations in Section 2.1 has been updated to make it consistent with the notation used in Chapter 3.

The cumulative nature of the thesis with self-contained chapters leads to some redundancy. For instance, Table 2.1 in Section 2.1 is replicated in Section 3.4 as Table 3.2.

The bibliography for all chapters is given at the end of the thesis.

CHAPTER 2

Motivation

2.1 Contextual Information[1]

2.1.1 Introduction

Most data in relational databases include numerical parts that must be analyzed. Since interesting analytical tasks often require both linear and relational operations, these operations must be combined to answer a single analytical query. However, linear operations are only defined and performed on numerical values and do not support the processing of the rich contextual information of relations. Our approach supports data analysis that require a mixture of relational and linear algebra operations.

Consider Figure 2.1 with relations s, f, and p from a plant bakery. Relation s records past, current, and planned future information about shift length (in hours) and number of workers per day. Relation f records information about the scrap percentage of different machines. Relation

[1] A version of this paper is published as *O. Dolmatova, N. Augsten, and M. Böhlen, "Preserving Contextual Information in Relational Matrix Operations.", Proceedings of the 36th International Conference on Data Engineering (ICDE), Dallas 2020, 4 pages*

p records information about the profit from the sales of items. For instance, tuple s_1 states that on March 21, 2019, four workers were working a ten hour shift. Tuple f_1 states that, on this day, the scrap of mixing machine MM1 was 1.2%. Tuple p_1 states that on March 21, 2019, 53 items of Moscow bread were sold with a rebate of 5% and a total profit of 6,100 dollars. Relation s is updated infrequently and includes planning information; relations f and p are updated regularly as new information about, respectively, scrap and profit becomes available.

As of January 2020, a revised work schedule with an increased number of workers and shorter shifts shall be put in place. Multiple linear regression [YS09], with independent variables L[2] and F, shall be used to predict the profit in 2020. To solve this type of analytical tasks we propose relational matrix operations that preserve contextual information. The preservation of contextual information is based on *splitting* relations into an *application part* (numerical data for matrix operations) and *contextual information* (giving meaning to the numerical data), which are processed independently during an operation, and *merging* the result of a linear operation with contextual information into the final result relation.

s (shift)

	Date	Length	Workers
	...	...	...
s_1	2019-03-21	10	4
s_2	2019-03-22	9	6
...	...	...	...
s_3	2020-01-01	5	15
	...	...	...

f (fault)

	Date	Machine	Fault
	...	...	...
f_1	2019-03-21	MM1	1.2
f_2	2019-03-21	BM1	0.3
f_3	2019-03-21	BM2	2.1
f_4	2019-03-22	BM1	2.5

p (profit)

	Date	Item	Rebate	Number	Profit
	...	...	...	...	...
p_1	2019-03-21	Moscow Bread	5	53	6,100
p_2	2019-03-21	Raisin Cake	7	70	11,900
p_3	2019-03-22	Salt Sticks	9	40	3,100
p_4	2019-03-22	Moscow Bread	2	120	12,200

Figure 2.1: Sample database

Current relational systems are poorly equipped to solve complex analytical queries since not even basic matrix operations are natively supported. There have been efforts to integrate linear algebra into the relational model. Such approaches introduced matrices as ordered data structures

[2]We use the first character of the attribute name to refer to attributes.

that are either used as attributes in relations [BDF+98, LGG+17], without the possibility to store contextual information for each matrix cell, or as standalone objects [SBPR11, ZKM13], which offer only matrix operations without relational database features.

As a running example, we consider the three key steps of predicting the profit: (1) data preparation, (2) computing the linear regression, (3) predicting the future profit. The linear equation for the regression has the form $F * x_F + L * x_L = P$, where coefficients x_F and x_L denote the (possibly negative) contribution of F and L (independent variables) to profit P (dependent variable). Figure 2.2 illustrates the profit estimation expressed as a sequence of relational operations[3] and relational matrix operations (`cpd`, `inv`, `mmu` denote relational crossproduct, inversion, and matrix multiplication, respectively). Note the seamless integration of matrix and relational algebra operators: The entire process frequently switches between linear and relational operations.

$$
\begin{aligned}
u1 &\leftarrow {}_{D}\vartheta_{AVG(F)\rightarrow F}(f) \bowtie \pi_{D,L}(s) \\
u2 &\leftarrow {}_{D}\vartheta_{SUM(P)\rightarrow P}(p)
\end{aligned}
\Big\} \textit{prepare data}
$$

$$
\begin{aligned}
u3 &\leftarrow \mathtt{cpd}^{V}_{D;D}(u1,u1) \\
u4 &\leftarrow \mathtt{inv}_{V}(u3) \\
u5 &\leftarrow \mathtt{cpd}^{V}_{D;D}(u1,u2) \\
u6 &\leftarrow \mathtt{mmu}_{V;V}(u4,u5)
\end{aligned}
\Big\} \textit{compute linear regression}
$$

$$
\begin{aligned}
u7 &\leftarrow \sigma_{Year(D)=2020}(s) \times \vartheta_{AVG(F)\rightarrow F}(f) \\
u8 &\leftarrow \mathtt{mmu}_{D;V}(u7,u6) \\
u9 &\leftarrow {}_{Month(D)}\vartheta_{SUM(P)\rightarrow P}(u8)
\end{aligned}
\Big\} \textit{predict profit}
$$

Figure 2.2: Algebra expression for profit estimation in 2020

The steps to estimate the profit are as follows: 1. Data preparation ($u1$, $u2$): Relational operations are used to compute the average daily scrap ($u1$) and the daily profit per shift ($u2$). 2. Linear regression ($u3$, $u4$, $u5$, $u6$): We use the ordinary least squares (OLS) method [RRT95, p. 25] to compute the linear regression. Thus, we compute $(\mathtt{CPD}(A,A))^{-1} * \mathtt{CPD}(A,V)$, where `CPD` denotes the matrix crossproduct. A is the matrix with the independent variables (i.e., $u1$), and V is the vector with the dependent variable (i.e., $u2$). The intermediate result relations are shown in Figure 2.3. 3. Profit estimation ($u7$, $u8$, $u9$): The coefficients defined by the linear regression are multiplied by the values of the corresponding independent variables (i.e., L and F) in 2020 to predict the future profit.

[3] We write $X \rightarrow A$ to rename attribute/expression X to A.

$u3$

V	F	L
F	7.8	34.5
L	34.5	181

$u4$

V	F	L
F	0.9	-0.17
L	-0.17	0.04

$u5$

V	P
F	59,850
L	317,700

$u6$

	V	P
z_1	F	-144
z_2	L	2533.5

Figure 2.3: Steps during the linear regression computation

Note that all operations operate on and return relations, and preserve contextual information. Consider crossproduct $u5 = \texttt{cpd}^{V}_{D;D}(u1,u2)$ from Figure 2.2. The rows of $u1$ and $u2$ are ordered according to the values of attribute D. The rest of the attributes in $u1$ and $u2$, i.e., attributes F and L from $u1$ and attribute P from $u2$, are used to calculate the crossproduct. Thus, the crossproduct is computed between the matrix consisting of the values of $u1.F$, $u1.L$ and the matrix of the values of $u2.P$, and the result relation includes attribute V with contextual information. The result of the crossproduct is relation $u5$ with schema (V,P). The values of V are the attribute names of the application part of $u1$. These values are essential to interpret the tuples in $u6$. For example, tuple z_1 states that the profit decreases by 144 dollars for each percent of scrap (the independent variable F).

Our approach is purely based on relations and does not introduce any ordered data structures. Instead, the relevant row order for matrix operations is computed from contextual information in the input relations. At the system level we evaluate our approach by integrating it into a column store. We extended the kernel of MonetDB with relational matrix operations implemented over binary association tables (BATs). The column nature of MonetDB supports the splitting of a relation into the application part and the contextual information, and the merging of a linear algebra result with the contextual information into a result relation.

Our contributions are as follows:

- We propose new *relational matrix operations that preserve contextual information*. This is the first approach that performs relational matrix operations and does not require ordered data structures.

- The relational matrix operations are *closed with respect to the relational model*, i.e., all operations are applied to relations and return relations as results.

- We show that our solution is practically feasible and able to *leverage existing data structures* by integrating it into MonetDB.

2.1.2 Context Preserving Relational Matrix Operations

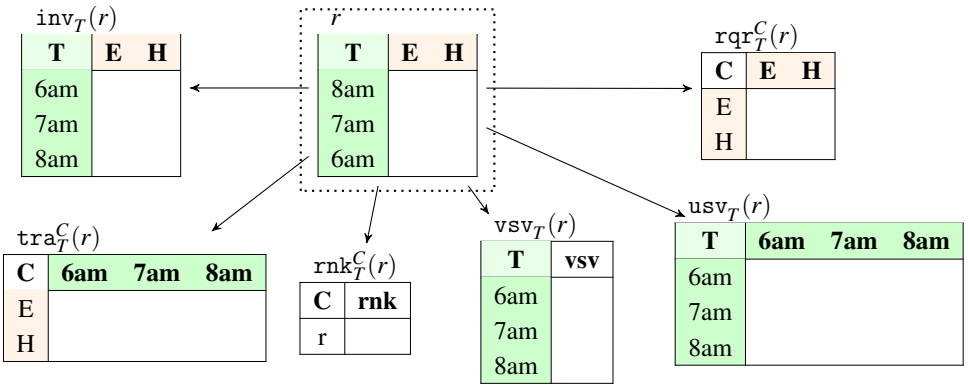
Notation: A relation r is a set of tuples r_i with schema $\mathscr{R}$. A schema $\mathscr{R} = (A, B, \ldots)$ is a finite, ordered set of attribute names. Ordered subsets of a schema, $\mathbf{U} \subseteq \mathscr{R}$, are typeset in bold, and $\overline{\mathbf{U}} = \mathscr{R} \setminus \mathbf{U}$ denotes the complement of a set of attributes in a relation schema. An $n \times k$ matrix m is a two-dimensional array with n rows and k columns. The *column cast* maps a set of attribute values into an ordered set by ordering them; the column cast of attribute $\mathbf{O}$ is denoted by $\triangledown\mathbf{O}$. The *schema cast* $\Delta\mathbf{U}$ creates a matrix m (with a single column) from the attribute names of schema $\mathbf{U}$, preserving the attribute order. The result of *concatenating* matrix d and matrix e with k rows each is a new matrix $h = d \square e$ with k rows, where each row is the concatenation of the corresponding rows from d and e.

We consider the matrix operations from the R Matrix Algebra [QR17]: Element-wise multiplication (`EMU`), matrix multiplication (`MMU`), outer product (`OPD`), cross product (`CPD`), matrix addition (`ADD`), matrix subtraction (`SUB`), transpose (`TRA`), solve equation (`SOL`), inversion (`INV`), eigenvectors (`EVC`), eigenvalues (`EVL`), QR decomposition (`QQR`, `RQR`), SVD – single value decomposition (`DSV`, `USV`, `VSV`), determinant (`DET`), rank (`RNK`), and Choleski factorization (`CHF`). Since QR and SVD return more than one matrix we split these operations (e.g., operations `QQR` and `RQR` for QR decomposition).

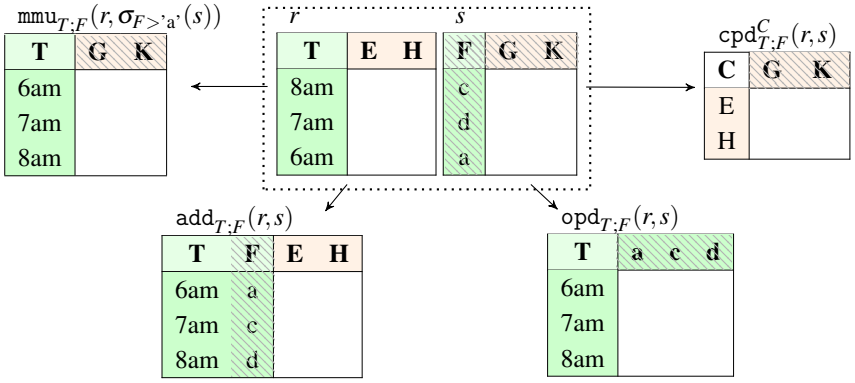
For each matrix operation we define how contextual information is preserved by the corresponding relational matrix operation. We use upper case for matrix operations (e.g., `TRA`) and lower case for relational matrix operations (e.g., `tra`). For each argument relation, r, of a relational matrix operation one parameter must be specified: The *order schema* $\mathbf{U}$ imposes an order on the tuples for this operation. The attributes of the order schema must form a key. The rest of the attributes, i.e., $\overline{\mathbf{U}} = \mathscr{R} \setminus \mathbf{U}$, is called the application schema and identifies the data to which the matrix operation is applied.

The order schema $\mathbf{U} \subseteq \mathscr{R}$ splits relation r into four non-overlapping areas: Application part $\pi_{\overline{\mathbf{U}}}(r)$; order part $\pi_{\mathbf{U}}(r)$; application schema $\overline{\mathbf{U}}$; and order schema $\mathbf{U}$. The parts of r that do not include numeric matrix values, i.e., the schemas and the order part, form the *contextual information* of r. Intuitively, the application schema provides context for columns while the order part and schema provide context for rows.

Example 6. Order schema $\mathbf{U} = (T)$ splits relation r in Figure 2.4 into four parts: Application part (white area); order part $(8am, 7am, 6am)$; order schema (T); and application schema $\overline{\mathbf{U}} = (E, H)$. □



(a) Unary Operations



(b) Binary Operations

Figure 2.4: Illustration of the preservation of contextual information

Matrix $(\mu, \overline{\mu})$ and relation (γ) constructors split and combine application part and context information to transition between matrices and relations without loosing relevant contextual information. At the implementation level, constructors are very efficient since they split and combine lists of attribute names and do not access the data (cf. Section 2.1.3).

Figure 2.4 illustrates the preservation of contextual information for relational matrix operations. The cardinality of the result matrix determines the contextual information that is inherited. For instance, `inv` preserves row and column contextual information, whereas `cpd` transforms column contextual information of the first argument relation to row contextual information in the result.

Table 2.1 defines how input relations must be split and how result matrices are merged to relations. All definitions preserve contextual information as illustrated in Figure 2.4. Consider relational matrix inversion $\texttt{inv}_{\mathbf{U}}(r)$ with order schema $\mathbf{U}$. $\mu_{\mathbf{U}}(r)$ are the rows of the order part,

$\overline{\mu}_{\mathbf{U}}(r)$ are the rows of the application part, $\texttt{INV}(\overline{\mu}_{\mathbf{U}}(r))$ is the result of matrix inversion, and $\mathbf{U} \circ \overline{\mathbf{U}}$ is the result schema.

$\texttt{inv}_{\mathbf{U}}(r) = \gamma(\mu_{\mathbf{U}}(r) \,\square\, \texttt{INV}(\overline{\mu}_{\mathbf{U}}(r)), \mathbf{U} \circ \overline{\mathbf{U}})$
$\texttt{usv}_{\mathbf{U}}(r) = \gamma(\mu_{\mathbf{U}}(r) \,\square\, \texttt{USV}(\overline{\mu}_{\mathbf{U}}(r)), \mathbf{U} \circ \triangledown\mathbf{U})$
$\texttt{vsv}_{\mathbf{U}}(r) = \gamma(\mu_{\mathbf{U}}(r) \,\square\, \texttt{VSV}(\overline{\mu}_{\mathbf{U}}(r)), \mathbf{U} \circ (\texttt{vsv}))$
$\texttt{add}_{\mathbf{U};\mathbf{V}}(r,s) = \gamma(\mu_{\mathbf{U}}(r) \,\square\, \mu_{\mathbf{V}}(s) \,\square\, \texttt{ADD}(\overline{\mu}_{\mathbf{U}}(r), \overline{\mu}_{\mathbf{V}}(s)), \mathbf{U} \circ \mathbf{V} \circ \overline{\mathbf{U}})$
$\texttt{opd}_{\mathbf{U};\mathbf{V}}(r,s) = \gamma(\mu_{\mathbf{U}}(r) \,\square\, \texttt{OPD}(\overline{\mu}_{\mathbf{U}}(r), \overline{\mu}_{\mathbf{V}}(s)), \mathbf{U} \circ \triangledown\mathbf{V})$
$\texttt{mmu}_{\mathbf{U};\mathbf{V}}(r,s) = \gamma(\mu_{\mathbf{U}}(r) \,\square\, \texttt{MMU}(\overline{\mu}_{\mathbf{U}}(r), \overline{\mu}_{\mathbf{V}}(s)), \mathbf{U} \circ \overline{\mathbf{V}})$
$\texttt{rnk}^{C}_{\mathbf{U}}(r) = \gamma(r \circ \texttt{RNK}(\overline{\mu}_{\mathbf{U}}(r)), (C, \texttt{rnk}))$
$\texttt{cpd}^{C}_{\mathbf{U};\mathbf{V}}(r,s) = \gamma(\Delta\overline{\mathbf{U}} \,\square\, \texttt{CPD}(\overline{\mu}_{\mathbf{U}}(r), \overline{\mu}_{\mathbf{V}}(s)), (C) \circ \overline{\mathbf{V}})$
$\texttt{rqr}^{C}_{\mathbf{U}}(r) = \gamma(\Delta\overline{\mathbf{U}} \,\square\, \texttt{RQR}(\overline{\mu}_{\mathbf{U}}(r)), (C) \circ \overline{\mathbf{U}})$
$\texttt{tra}^{C}_{\mathbf{U}}(r) = \gamma(\Delta\overline{\mathbf{U}} \,\square\, \texttt{TRA}(\overline{\mu}_{\mathbf{U}}(r)), (C) \circ \triangledown\mathbf{U})$

Table 2.1: Splitting and morphing relations and matrices

Operations that yield relations with a different number of rows than any of the input relations add a new attribute C with contextual information to the result relation. Depending on the operation, the values of attribute C are the attribute names of the application schema of one of the input relations, or the name of the input relation.

2.1.3 Implementation and SQL Extension

MonetDB Integration

MonetDB stores each column of a table as a binary association table (BAT). A BAT is a table with a head and tail. The head is a column with object identifiers (OID), while the tail is a column with attribute values. B, X, and Y denote lists of BATs, and we use indices, e.g., Y_1, to refer to an individual BAT. All attribute values of a tuple in a relation have the same OID value. MonetDB operations manipulate BATs and all operations are represented and executed as sequences of BAT operations. An example BAT operation is $B_1 * B_2$, which is the element-wise multiplication. BAT operation $\downarrow$ sorts the OIDs of one BAT according to the order of the OIDs of another BAT from the same relation. For instance, $X \downarrow Y$ returns BAT X, whose OIDs have the same order as the OIDs of BAT Y. $X \downarrow X$ denotes X sorted by its attribute values.

A relational matrix operation is processed in the following five steps: *1) Splitting* divides a relation into two parts; *2) Sorting* determines the order of the tuples for the matrix operation; *3) Morphing* transforms contextual information according to the operation; *4) Compute* performs the matrix operation on the values of the application part; *5) Merging* combines the result of the matrix operation with contextual information and constructs the result relation. Note that splitting and merging correspond to the matrix and relation constructor, respectively, and work at the schema level only.

Algorithm 1: `inv`(**U**, **O**, r)

```
1  B ← BATs(r) ;
2  C ← {} ; D ← {} ; Y ← {} ;
3  for b ∈ B do
4      if b ∈ U then D ← D ∪ b;                                   // μ_U(r)
5      else C ← C ∪ b;                                            // μ̄_U(r)
6  X ← sort(D) ;
7  for b ∈ C do Y ← Y ∪ b↓X;
8  n ← Y.length ;                                                 // INV(Y)
9  H ← IDmatrix(n);
10 for i = 1 to n do
11     v1 ← sel(Y_i, i);
12     Y_i ← Y_i / v1;
13     H_i ← H_i / v1;
14     for j = 1 to n do
15         if i ≠ j then
16             v2 ← sel(Y_j, i);
17             Y_j ← Y_j − Y_i * v2;
18             H_j ← H_j − H_i * v2;
19 Z ← Merge(X, H) ;                                              // γ(X □ H, U ∘ Ū)
20 return Z;
```

Algorithm 1 illustrates the five steps for the relational matrix operation $\texttt{inv}_{\mathbf{U}}(r)$. Lines 3-7 correspond to the two matrix constructors from the definition of $\texttt{inv}_{\mathbf{U}}(r)$ in Table 2.1: $\mu_{\mathbf{U}}(r)$ and $\overline{\mu}_{\mathbf{U}}(r)$. The BATs of relation r are split, sorted, and morphed to get BATs X with the order part and BATs Y with the application part. Lines 8-18 illustrate the Gauss Jordan elimination for the $\texttt{INV}(\overline{\mu}_{\mathbf{U}}(r))$ computation. Function $IDmatrix(n)$ creates a list of BATs that represents an

identity matrix of size $n \times n$. The selection operation $sel(B_1, i)$ returns the i^{th} value in B_1. In line 19, the relation constructor combines the result relation by merging the morphed order part and the result application part.

The algorithm is representative for the other relational matrix operations and illustrates the elegance and simplicity of the integration of our solution into MonetDB.

SQL Extension for Relational Matrix Operations

The relational matrix operations have been made available in the FROM clause of SQL as, respectively, unary and binary operations with the names in Table 2.1. Each argument relation r is defined through an extended SQL statement that allows to specify the order schema as follows:

```
SELECT * FROM r BY U
```

Figure 2.5 illustrates the syntax extension for the algebra expression in Figure 2.2. The relational matrix operations and the extension to specify application schema and ordering are highlighted in red.

```
1  WITH
2    t1(F, D) AS ( SELECT AVG(F) AS F, D FROM fault GROUP BY D ),
3    u1(D, F, L) AS ( SELECT t1.D, F, L FROM t1 NATURAL JOIN s )
4
5  SELECT SUM(P) AS P
6  FROM MMU ( SELECT D, F, L
7             FROM shift, ( SELECT AVG(F) AS F FROM fault ) t2
8             WHERE YEAR(D) = 2020
9             BY D,                                                    u7
10            SELECT *
11            FROM MMU ( SELECT *
12                       FROM INV ( SELECT *
13                                  FROM CPD[V] (SELECT * FROM u1 BY D,
14                                               SELECT * FROM u1 BY D )
15                                  BY V )
16                       BY V,                                          u4
17                       SELECT *
18                       FROM CPD[V] ( SELECT * FROM u1 BY D,
19                                     SELECT D, SUM(P) AS P
20                                     FROM profit
21                                     GROUP BY D
22                                     BY D )
23                       BY D )                                         u5
24            BY V )                                                  u6
25 GROUP BY MONTH(D);
```

Figure 2.5: SQL statement that is equivalent to the algebra expression in Figure 2.2

2.2 Shape Preserving Iterations[4]

2.2.1 Introduction

In the era of big data analytics many researchers have to deal with constantly growing data sets that must be analyzed with state-of-the-art data science methods based on iterative methods. *Propensity score matching* is a statistical technique that estimates the effect of medical interventions, e.g., medical treatments. It reduces the bias that exists if we simply compare outcomes among patients that received the treatment versus those that did not. A propensity score is associated with each tuple and can be used to build patient cohorts with comparable patients, i.e., patients with a similar propensity score. The approximation of the propensity score is calculated with the gradient descent method that uses fixed point computations and iteratively refines initially guessed values. Since neither gradient descent nor fixed point iterations are available in SQL, applications must export, transform, and import data into statistical analysis environments [Pro19, PVG+11] to compute propensity scores.

Our goal is to enable propensity score matching computation over data that is stored in relations. Towards this goal, we extend SQL with *shape preserving iterations* that include an iteration body, an exit condition, and that iterate over a relation of a constant size (i.e., the number of tuples and attributes does not change). A shape preserving iteration repeat its iteration body computation until the exit condition evaluates to true. Shape preserving iterations support methods that repeatedly refine a set of values until a fixed point is reached and, thus, permit in-database propensity score calculation. Shape preserving iterations start with a *randomly initialized relation*. We show how to randomly initialize a relation that has the required schema and tuples. One important property of shape preserving iterations is that they iterate over and return relations with *contextual information* [DAB20b, DAB20a]. Contextual information guarantees that each relation has a proper schema and includes at least one attribute whose values describe and identify tuples in this relation. We show the feasibility of our solution by implementing propensity score computation in MonetDB and conducting an experimental evaluation.

Our technical contributions are the following:

[4]A version of this paper is published as *M. Böhlen, O. Dolmatova, M. Krauthammer, A. Mariyagnanaseelan, J. Stahl and T. Surbeck, "Iterations and Propensity Score Matching in MonetDB.", Advances in Databases and Information Systems, 24th East European Conference (ADBIS), Lyon 2020, 14 pages*

- We identify *shape preserving iterations* as a new type of iterations required for fixed point computations over relations. We offer an SQL extension to do in-database propensity score matching.
- We show how to create an input relation with contextual information for the gradient descent computation using *randomly initialized relations*.
- We confirm the feasibility of our approach by empirically comparing the runtime of our solution with a native MonetDB implementation that flattens loops.

The section is organized as follows. Subsection 2.2.2 describes our application scenario. We introduce basic terminology in Subsection 2.2.3. We introduce our approach and discuss the extensions required for propensity score computations in Subsection 2.2.4. Subsection 2.2.5 describes the implementation in MonetDB and Subsection 2.2.6 evaluates our solution.

2.2.2 Application Scenario

Figure 2.6 illustrates a sample of a data set with health information of patients. This is an extract from the right heart catheterization test data set [HJ19] with 63 attributes and 5735 patient records. For example, tuple $t1$ in relation *rhc* refers to the patient whose patient ID is 394, age is 67.7 years, weight is 65.9 kg, and blood pressure is 125.0 mm Hg. The patient has not received right heart catheter treatment (value of attribute *Treatment* is zero) and the outcome was positive (value of attribute *Death* is zero).

rhc

	PatientID	**Age**	**Weight**	**BloodPressure**	**Treatment**	**Death**
$t1$	394	67.7	65.9	125.0	0	0
$t2$	979	69.7	54.0	58.0	0	1
$t3$	1198	65.6	75.7	45.0	1	1
$t4$	4314	68.5	94.1	55.0	0	1

Figure 2.6: Excerpt of the right heart catheterization data set

The task is to group patients into cohorts, such that each cohort consists of comparable patients. Comparable cohorts allow to compute the true effect of a treatment and decide whether the treatment is successful or not. Treatment success analysis is a common approach to predict the recovery outcome of a medication. Data sets processed in treatment success analysis typically

include a binary feature variable that indicates whether a patient received treatment. For example, tuple $t3$ in relation *rhc* represents a patient who received treatment (value of attribute $Treatment$ is 1). Intuitively, one is tempted to divide the input data into a set of patients having received treatment and compare their outcome of recovery to the set of untreated patients. However, this strategy ignores any biases in the data [RR83]. It is likely that among the patients there are some who would have recovered anyway because of their general health condition, but who still received the treatment. Similarly, it is not enough to separate patients by a recorded feature, such as gender, because groups of male and female patients might be incomparable due to other differences.

Treatment success analysis requires unbiased data, i.e., a data set containing treated and untreated patients, which according to their conditions (i.e., feature values) are comparable. Since most of the data is historical, there is no possibility to randomly assign treatment to patients. Instead, comparable patients must be selected deliberately to form cohorts. The propensity score represents the impact of all characteristic to a treatment and, thus, allows to match patients with similar scores. Thus, the propensity score of a patient is the probability of getting treatment. Typically, the distribution into groups is done via *Propensity Score Matching*. We use shape preserving iterations to integrate propensity score matching into SQL and MonetDB, and we discuss the details of our solution of the application scenario in Section 2.2.4.

2.2.3 Background

We leverage the extension of SQL with relational matrix operations that supports basic matrix operations, such as multiplication and inversion, over relations [DAB20b, DAB20a]. Each relation is divided into two parts as illustrated in Figure 2.7: contextual information and the application part. The gray cells are the contextual information, and the white cells are the application part. The contextual information identifies and describes each cell in the application part. For example, value 125.0 in relation $t1$ is the blood pressure of the patient with ID 394. The application part is used in matrix operations. Each relational matrix operation takes one or two relations with contextual information as input and yields a result relation with contextual information. The contextual information of the result relation is inherited from the contextual information of the input relation and is responsible for identifying and describing tuples and attributes. The inheritance is based on the shape of the result relation.

$t1$

P	A	B	W
394	67.7	125.0	65.9
979	69.7	58.0	54.0
1198	65.6	45.0	75.7
4314	68.5	55.0	94.1

$t2$

P	T
394	0
979	0
1198	1
4314	0

$v = \mathtt{cpd}^F(t1,t2)$

F	T
A	65.6
B	45.0
W	75.7

Figure 2.7: Input and result relations for cpd

In the SQL extension each input relation r is followed by a list of attributes **U** that is the *order schema*. The order schema is a part of the contextual information and determines the order of tuples for the purpose of a matrix operation. The rest of the attributes in r, $\mathscr{R} \setminus \mathbf{U}$, is the *application schema*. For example, the binary relational matrix multiplication is expressed as shown on Figure 2.8.

```
1  SELECT * FROM MMU( r BY U, s BY V );
```

Figure 2.8: Relational matrix multiplication

Here, r and s are input relations, and **U** and **V** are order schemas that determine the order of tuples for the multiplication. The relational matrix multiplication corresponds to matrix multiplication $a * b$, where a and b are matrices composed of attributes $\mathscr{R} \setminus \mathbf{U}$ of relation r ordered by **U** and attributes $\mathscr{S} \setminus \mathbf{V}$ from relation s ordered by **V**, respectively.

Example 7. Figure 2.9 shows the computation of the relational matrix crossproduct (cpd) between attributes A, B, W and attribute T from relation *rhc* sorted by the values of attribute P.

```
1  SELECT *
2  FROM CPD[F]( ( SELECT P, A, B, W FROM rhc ) AS t1 BY P,
3               ( SELECT P, T FROM rhc ) AS t2 BY P
4             ) AS v;
```

Figure 2.9: Relational matrix crossproduct

Figure 2.7 illustrates the input and result relations of the crossproduct computation. Both subselects include attribute P to sort the tuples in $t1$ and $t2$ for the purpose of the cpd operation.

This relational matrix crossproduct corresponds to matrix crossproduct $a^t * b$, where a is the matrix composed of attributes A, B, and W of relation *rhc* ordered by P and b is the matrix composed of attribute T from relation *rhc* ordered by P. Note that result relation v includes contextual information together with the numeric result of the crossproduct computation. Relation v has schema

(F, T). Attribute F, whose name is specified with the superscript after the operation, includes inherited attribute names A, B, and W from $t1$. Attribute T includes the numeric result and inherits its name from $t2$ [DAB20b]. The inheritance of these values is based on the cardinalities of the result matrix of the crossproduct. □

2.2.4 Propensity Score

Propensity score matching builds cohorts based on the estimation of the propensity score. Propensity score matching requires to perform the following steps: (1) computation of gradient descent between features and a target, e.g., between attributes A, B, and W as features and attribute T as a target from relation *rhc*; (2) estimation of the propensity score by multiplying features and the coefficients from the gradient descent; and (3) grouping of estimated propensity scores according to their similarity. We consider these steps in the following subsections.

Gradient Descent and Shape Preserving Iterations

Gradient descent [Rud16] is an algorithm that is often used for classification tasks, such as logistic regression. Gradient descent is an approximation method, where a cost function is iteratively minimized, while letting the coefficients converge to the optimum for the given data set [MJ09].

In the context of relations, gradient descent takes three argument relations: The first relation includes feature attributes, the second relation includes an attribute that represents the target, the third relation includes an attribute with the initially guessed coefficients. The last input relation is also the output relation and its values (i.e., coefficients) are iteratively refined until the coefficients have converged to the real impact of the features on the target. The iteration used in gradient descent has two key properties: (1) it is a fixed point iteration with a cost function that must be minimized; (2) the shape of the iterated result relation remains the same (i.e., the iteration refines values, but does not change the number of attributes or tuples). We term such iterations *shape preserving iterations*.

Shape preserving iterations are used in iterative methods [Var62] from numerical analysis that refine matrices with randomly initialized values. Iterative methods are used to solve problems for which direct methods are very expensive or do not exist, such as logistic regression.

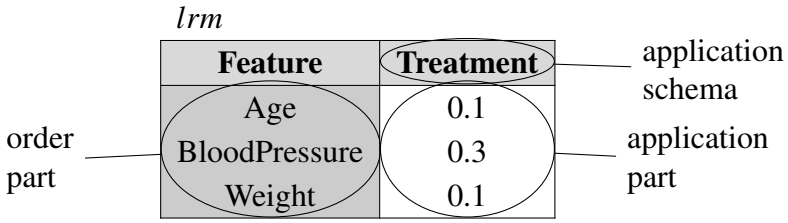


Figure 2.10: Structure of an iterated relation

The division of a relation into contextual information and an application part [DAB20a] is satisfied for shape preserving iterations. Consider relation *lrm* in Figure 2.10. The relation quantifies for each feature its impact on the treatment. The gray cells are the contextual information and remain unchanged during the iteration. The contextual information determines the shape of the relation. The white cells are the application part with the values that are refined during an iteration.

Randomly Initialized Relations

Gradient descent takes a relation with guessed coefficients as input. The start point is a *randomly initialized relation*, i.e., a relation with contextual information and an application part over which the iteration is performed to approximate the solution. The values in the application part of a randomly initialized relation are generated according to the provided distribution. The contextual information is inherited from existing relations.

Relation *lrm* in Figure 2.10 is the start point for the gradient descent of our application scenario. It is initialized as illustrated in Figure 2.11 (see also Figure 2.7).

```
1  SELECT F, uniform [0, 1] AS T
2  FROM CPD[F]( ( SELECT P, A, B, W FROM rhc ) AS t1 BY P,
3               ( SELECT P, T FROM rhc ) AS t2 BY P );
```

Figure 2.11: SQL query for random initialization

Relation *lrm* inherits its contextual information from relations $t1$ and $t2$. The shape and the contextual information of *lrm* are determined by operation `cpd`. The contextual information of *lrm* includes the values of attribute F. By definition of `cpd`, they are the names of attributes A, B, and W, and they describe feature coefficients. The application schema of *lrm* is attribute T. It is inherited from the application schema of $t2$. The application part of *lrm* (i.e., values of attribute T) includes random values uniformly distributed between 0 and 1. In order to randomly

initialize a relation, the appropriate relational matrix operation is applied to existing relations. The operation provides contextual information, i.e., a skeleton for the application part. The values in the application part are independent of the operation result.

Gradient Descent with Iterations in SQL

We use the `WITH` clause to perform gradient descent with shape preserving iterations. Figure 2.12 illustrates the syntax.

```
1 WITH
2   ITERATED r
3     INITIAL (R)
4     AS (Q)
5   UNTIL P
6 SELECT * FROM r;
```

Figure 2.12: Iterations in SQL

Relation *r* is a randomly initialized relation initialized with query `R`. Query `Q` corresponds to the iteration body. Values in relation *r* are updated with the result values yielded by *Q*. Predicate `P` specifies the exit condition. Thus, `Q` is repeated and relation *r* is updated until `P` evaluates to true.

Figure 2.13 illustrates the SQL statement for gradient descent computation of our application scenario. The gradient descent is computed over attributes *A*, *B*, and *W* as features, and attribute *T* as a target from relation *rhc*. Since attribute *T* includes binary values, the gradient descent is based on standard logistic regression. The SQL statement corresponds to a standard gradient descent algorithm [Rud16], where α is the stepsize, and τ is the threshold. The iteration part of the query is framed and consists of `R`, `Q`, and `P`.

The gradient descent is performed over the randomly initialized relation *lrm* that is initialized with subquery `R`. The values in attribute *F* remain unchanged, and the values in attribute *T* are refined during the computation. Expression 1/(1+1/EXP(T)) corresponds to the sigmoid function $\frac{1}{1+e^{-T}}$. Since the sigmoid function takes real values and maps them into a range from zero to one, it is used in gradient descents for logistic regressions.

Subquery `Q` corresponds to the iteration body of the gradient descent. `cpd` is performed on lines 16-30 between features from relation *rhc* and the normalized error from relation *d*. Relation *d* is computed as a normalized difference between estimated target values (i.e., relation *e*) and real target values. `cpd` yields relation $g(F,T)$: Attribute *T* is the gradient calculated for the current coefficients. Line 13 states how relation *lrm* is updated after each iteration: The values

```
1  CREATE VIEW lr_coeff AS
2  WITH
3    prhc(A, B, W, P) AS (
4      SELECT A, B, W, P
5      FROM rhc ),
6    ITERATED lrm(F, T)
7      INITIAL (
8        SELECT F, uniform [0, 1] AS T
9        FROM CPD[F]( ( SELECT P, A, B, W FROM rhc ) AS t1 BY P,
10                    ( SELECT P, T FROM rhc ) AS t2 BY P )
11     )                                                          R
12     AS (
13       SELECT lrm.F, lrm.T - α*g.T
14       FROM lrm
15            JOIN
16            CPD[F]( prhc BY P,
17                    ( SELECT t1.P, t1.T/t2.N AS T
18                      FROM ( SELECT rhc.P, rhc.T - e.T AS T
19                             FROM rhc
20                                  JOIN
21                                  ( SELECT P, 1/(1+1/EXP(T)) AS T
22                                    FROM MMU ( prhc BY P, lrm BY F )
23                                  ) AS e
24                                  ON rhc.P = e.P
25                           ) AS t1,
26                           ( SELECT COUNT(*) AS N
27                             FROM rhc
28                           ) AS t2
29                    ) AS d BY P
30                  ) AS g ON lrm.F = g.F
31     )                                                          Q
32   UNTIL
33     ( SELECT SUM(-rhc.T*log(e.T)-(1-rhc.T)*log(1-e.T))
34       FROM rhc
35            JOIN
36            ( SELECT P, 1/(1+1/EXP(T)) AS T
37              FROM MMU ( prhc BY P, lrm BY F )
38            ) AS e
39            ON rhc.P = e.P
40     )
41      <
42     ( SELECT τ * COUNT(*)
43       FROM rhc )                                               P
44 SELECT * FROM lrm;
```

Figure 2.13: Gradient descent over relation *rhc*

in attribute F are preserved, and the gradient is multiplied by the stepsize α and subtracted from the current coefficients.

Subquery `P` corresponds to the exit condition. It determines whether the cost function calculated between the real target values $rhc.T$ and the estimated target values $e.T$ is below the given threshold t.

The relational matrix operations are highlighted with red color, and the new constructs for the iteration are highlighted with green color. The result of the calculation of the SQL query is view lr_coeff that contains the logistic regression coefficients.

Figure 2.14 illustrates relation *lrm* during the computation of the gradient descent for our application scenario: Input (lrm_0), intermediate (lrm_{50}, lrm_{100}, lrm_{200}, lrm_{250}), and result (lrm_{266}) relations. The subscript denotes the iteration step, in which the relation is computed. The intermediate relations show how the coefficients converge during the gradient descent. For example, the coefficient for *W* converges in the beginning while the coefficient for *A* converges towards the end of the computation.

lrm_0

F	T
A	0.1
B	0.3
W	0.1

lrm_{50}

F	T
A	0.00
B	-0.06
W	0.03

lrm_{100}

F	T
A	0.03
B	-0.10
W	0.03

lrm_{200}

F	T
A	0.08
B	-0.16
W	0.03

lrm_{250}

F	T
A	0.10
B	-0.19
W	0.03

lrm_{266}

F	T
A	0.11
B	-0.20
W	0.03

Figure 2.14: Gradient descent: Input, intermediate, and output relations

Estimation and Matching

The propensity score is estimated by multiplying the features and the logistic regression coefficients from lr_coeff. After the estimation, groups of propensity scores are formed. Grouping data into percentiles according to propensity scores reduces bias and allows for a proper treatment success analysis [RR83]. Forming groups of patients with similar propensity scores is accomplished by matching tuples. There exist several approaches to perform the matching, such as stratification matching and caliper matching [Aus11, SGC07]. For stratification matching, the range of propensity scores is split into equally sized buckets, and each tuple is assigned to a bucket. Each bucket holds comparable tuples with treated and untreated patients.

In Figure 2.15 we estimate the propensity score for patients with relational matrix multiplication between features in *rhc* and coefficients in lr_coeff.

Figure 2.17 shows the result of the multiplication: Relation $propensity\_score$ has attribute *S* with an estimate of the probability for a treatment, i.e., the propensity score.

```
1  CREATE VIEW propensity_score AS
2  SELECT P, 1/(1+1/EXP(T)) AS S
3  FROM MMU ( ( SELECT A, B, W, P
4               FROM rhc ) AS t BY P,
5             lr_coeff BY F );
```

Figure 2.15: Estimating propensity score with the result of gradient descent

After the propensity score is calculated, the final step is a stratified matching applied to the estimated scores. Figure 2.16 shows the query that distributes the propensity scores into equally sized buckets of size 0.2.

```
1  SELECT P, CAST(S * 10 / 2 AS INT) AS I, S, T
2  FROM propensity_score NATURAL JOIN rhc;
```

Figure 2.16: Stratification matching of the propensity scores

Figure 2.17 illustrates the final result of propensity score matching with the stratification approach: Attribute *I* from relation *stratification_matching* states the bucket id for each patient.

propensity_score

P	S
394	0.000
979	0.091
1198	0.617
4314	0.345

stratification_matching

P	I	S	T
394	0	0.000	0
979	0	0.091	0
1198	3	0.617	1
4314	1	0.345	0

Figure 2.17: Propensity score: Estimation and matching

2.2.5 System Implementation in MonetDB

In this section we discuss the integration of gradient descent and shape preserving iterations into MonetDB. MonetDB is a column-store DBMS, which offers several routines optimized for column-oriented operations. It stores attributes of relations in *binary association tables* (BATs). A BAT consists of two arrays: One stores the attribute values and the other the object identifier (OID) for each tuple. Each relational operation is represented and executed as a sequence of BAT operations. MonetDB builds a *statement tree* where each node refers to one or more attributes, or the result of an operation on attributes.

Implementation

We implement gradient descent as a new node in the statement tree and a new BAT operation in MonetDB. We compare our implementation with the native MonetDB implementation where the number of iterations is predefined and iterations are flattened in the statement tree.

Node implementation In the node implementation we add a gradient descent node to a statement tree and a BAT operation with the gradient descent algorithm.

Algorithm 2: Gradient Descent

Input: BAT C (coefficients, initial guess),
list of BATs **A** = (A_1, A_2, ...) (feature vectors),
BAT T (target vector),
double α (stepsize),
double t (error threshold)
Output: BAT C (optimized coefficients)

```
1  n ← T.length()
2  repeat
3  |  E ← fill(0,n)
4  |  G ← []
5  |  foreach A_i ∈ A, val ∈C do
6  |  |  E ← E + val· A_i
7  |  D ← (sigmoid(E) − T)/n
8  |  foreach A_i ∈ A do
9  |  |  append(G, A_i· D )
10 |  C ← C - α· G
11 until cost(C, A, T) < t;
12 return C
```

Algorithm 2 illustrates our implementation of the new BAT operation. The gradient descent is applied to BATs. The algorithm takes as input BAT C with the initial coefficients, list of feature BATs **A**, target BAT T, gradient descent stepsize α, and error threshold t. The cost function in Equation 2.1 is the function we minimize with the gradient descent algorithm.

$$\texttt{cost}(\mathrm{C},\mathbf{A},\mathrm{T}) = (-\mathrm{T}\cdot\log(\texttt{sigmoid}(\mathbf{A}\cdot\mathrm{C})) - (1-\mathrm{T})\cdot\log(1-\texttt{sigmoid}(\mathbf{A}\cdot\mathrm{C})))/n \quad (2.1)$$

The algorithm returns the updated BAT C with the final coefficients as soon as the exit condition is reached, i.e., the error is smaller than the threshold.

First, BAT E, which is the non-normalized estimation of target T, is filled with zeros. Then the estimations based on current coefficients C are calculated in lines 5-6. Second, BAT D with the difference of the normalized estimation and the actual target is computed on line 7. BAT G with the current gradient is calculated on lines 8-9. BAT C witht the coefficients is updated on line 10.

Example 8. Consider Figure 2.18. It illustrates the first iteration step of Algorithm 2 performed over randomly initialized relation *lrm* shown in Figure 2.10 and relation *rhc*.

C
0.1
0.3
0.1

A_1	A_2	A_3
67.7	125.0	65.9
69.7	58.0	54.0
65.6	45.0	75.7
68.5	55.0	94.1

T
0
0
1
0

E
50.86
29.77
27.63
32.76

sigmoid(E)
1
1
1
1

D
0.25
0.25
0
0.25

G
51.45
59.50
53.50

C
0.05
0.24
0.05

Figure 2.18: The first step of the iteration

BAT C corresponds to attribute T from relation *lrm*, BAT list $\mathbf{A}$ = (A_1, A_2, A_3) corresponds to attributes A, B, W from relation *rhc*, BAT T corresponds to attribute T from relation *rhc*, stepsize α is 0.001, and threshold t is 0.25. BAT E (i.e., the current estimation of the target) is calculated by multiplying $\mathbf{A}$ with C, and the sigmoid function is applied to E. Then BAT D is calculated between the real target T and its estimation *sigmoid*(E). Next, each feature is multiplied with the normalized difference delivering gradient BAT G. Finally, BAT C is updated based on the values in G and the stepsize. After that, the cost function is applied to the refined coefficients C and the predicate is evaluated. □

Native MonetDB implementation Algorithm 2 is translated to a flattened statement tree where the number of iterations is predefined and the cost function is omitted. Thus, the iteration subtree Q is replicated multiple times.

2.2.6 Evaluation

We extended MonetDB v11.23.13 with the node implementation and the native implementation. We ran the evaluation on a virtual machine in the UZH ScienceCloud [Uni20] with Ubuntu 18.04.3 LTS, 2.593GHz Intel Haswell 4 CPU, and 16GB of RAM. Both server and client are running on the same machine.

We use synthetic data for the evaluation. Synthetic data is generated with function `make_classification` [BLB+13], which is a part of the Python library `scikit` [PVG+11]. All features in the generated data sets are informative, i.e., all features affect the target. All preconditions that are used for the generation of regression problems with different numbers of features and tuples are the same.

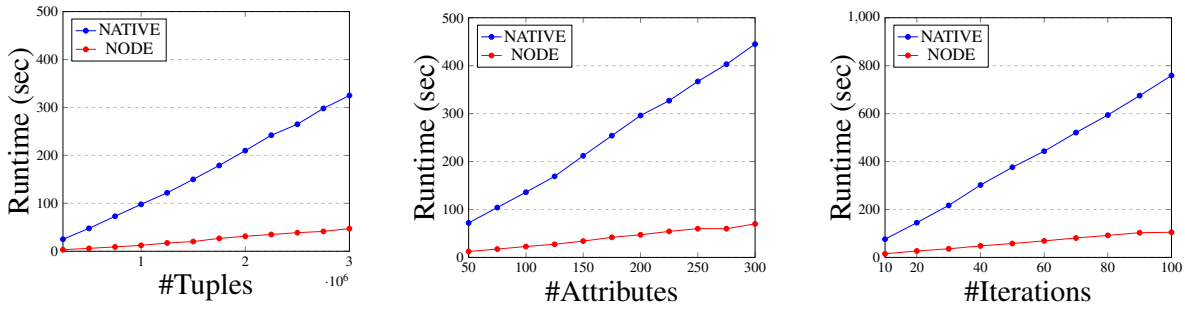


Figure 2.19: Runtime of gradient descent for varying number of tuples, attributes, and iterations

We perform gradient descent with the node and native implementations over relations of different sizes. We ensure that the node implementation performs as many iterations as the native implementation. For the native implementation we fixed the number of iterations by passing this number within the query. For the node implementation we set the tolerance to zero and additionally pass the maximal number of iterations. This guarantees that both approaches perform the same number of iterations.

Figure 2.19 illustrates the runtimes of both implementations. The left plot shows the runtimes for gradient descent applied to relations with 150 attributes (i.e., features) and a varying number of tuples. The plot in the middle shows runtimes for relations with 2,000,000 tuples and a varying number of attributes. Both evaluation runs were carried out with 30 iterations. The right plot illustrates the runtimes for relations with 150 attributes and 2,000,000 tuples, but with a varying number of iterations.

The node implementation shows better performance in all cases. Note that the node implementation computes the cost function after each iteration, while the native implementation performs only the iteration body. Since the native implementation flattens the statement tree, it creates huge trees with thousands of nodes, and thus, does not scale. Additionally, the native approach is not robust in terms of accuracy of the result because of the inability to access and evaluate the intermediate results during the tree creation.

CHAPTER 3

Building a System with Relational and Matrix Operations[1]

3.1 Introduction

A lot of data that is stored in relational databases includes numerical parts that must be analyzed, for example, sensor data from industrial plants, scientific observations, or point of sales data. The analysis of this data, which is not purely numerical but also includes important non-numerical values, demands mixed queries that apply relational and linear algebra operations on the same data.

Dealing with mixed workloads is challenging since the gap between relations and matrices must be bridged. Current relational systems are poorly equipped for this task. Previous attempts to deal with mixed workloads have focused on the implementation level, for example, by introducing ordered data types; by storing matrices in special relations or key-value structures; or by splitting queries into their relational and matrix parts. This work resolves the gap between relations and matrices.

[1] A version of this paper is published as *O. Dolmatova, N. Augsten, and M. Böhlen, "A Relational Matrix Algebra and its Implementation in a Column Store.", International Conference on Management of Data (SIGMOD), Portland 2020, 14 pages*

We propose a principled solution for mixed workloads and introduce the *relational matrix algebra* (RMA) to support complex data analysis within the relational model. The goal is to (1) solve the integration of relations and linear algebra at the logical level, (2) so achieve independence from the implementation at the physical level, and (3) prove the feasibility of our model by extending an existing system. We are the first to achieve these goals: Other works focus on facilitating the transition between the relational and the linear algebra model. We eliminate the dichotomy between matrices and relations by seamlessly integrating linear algebra into the relational model. Our implementation of RMA in MonetDB shows the feasibility of our approach.

We define linear operations over relations and systematically process and maintain non-numerical information. We show that the relational model is well-suited for complex data analysis if ordering and contextual information are dealt with properly. RMA is purely based on relations and does not introduce any ordered data structures. Instead, the relevant row order for matrix operations is computed from *contextual information* in the argument relations. All relational matrix operations return relations with *origins*. Origins are constructed from the contextual information (attribute names and non-numerical values) of the input relations and uniquely identify and describe each cell in the result relation.

We extend the syntax of SQL to support relational matrix operations. As an example, consider a relation *rating* with schema $(User, Balto, Heat, Net)$, that stores users and their ratings for the three films ("Balto", "Heat", and "Net", one column per film). The SQL query orders the

```
1  SELECT * FROM INV(rating BY User);
```

relation by users and computes the inversion of the matrix formed by the values of the ordered numerical columns. The result is a relation with the same schema: The values of attribute $User$ are preserved, and the values of the remaining three attributes are provided by matrix inversion (see Section 3.5 for details). The origin of a numerical result value is given by the user name in its table row and the attribute name of its column.

At the system level, we have integrated our solution into MonetDB. Specifically, we extended the kernel with relational matrix operations implemented over binary association tables (BATs). The physical implementation of matrix operations is flexible and may be transparently delegated to specialized libraries that leverage the underlying hardware (e.g., MKL [Int20] for CPUs or cuBLAS [NVI19] for GPUs). The new functionality is introduced without changing the main data structures and the processing pipeline of MonetDB, and without affecting existing functionality.

Our technical contributions are as follows:

- We propose the *relational matrix algebra* (RMA), which extends the relational model with matrix operations. This is the first approach to show that the relational model is sufficient to support matrix operations. The new set of operations is *closed*: All relational matrix operations are performed on relations and result in relations, and no additional data structure is required.
- We show that matrix operations are *shape restricted*, which allows us to systematically define the results of matrix operations over relations. We define *row and column origins*, the part of contextual information that describes values in the result relation, and prove that all our operations return relations with origins.
- We implement and evaluate our solution in detail. We show that our solution is feasible and leverages existing data structures and optimizations.

RMA opens new opportunities for advanced data analytics that combine relational and linear algebra functionality, speeds up analytical queries, and triggers the development of new logical and physical optimization techniques.

The chapter is organized as follows. Section 3.2 discusses related work. We introduce basics in Section 3.3 and introduce the relational matrix algebra (RMA) in Section 3.4. We show an application example in Section 3.5, discuss important properties of RMA in Section 3.6, and its implementation in MonetDB in Section 3.7. We evaluate our solution in Section 3.8.

3.2 Related Work

Relational DBMSs offer simple linear algebra operations, such as the pair-wise addition of attribute values in a relation. Some operations, e.g., matrix[2] multiplication, can be expressed via syntactically complex and slow SQL queries. The set of operations is limited and does not include operations whose results depend on the row order. For instance, there are no SQL solutions for inversion or determinant computation. Complex operations must be programmed as UDFs. Ordonez et al. [Ord07] suggest UDFs for linear regression with a matrix-like result type.

[2]Some approaches support multi-dimensional arrays. Since we target linear algebra, we focus on two dimensions and use the term *matrix* throughout.

The UTL_NLA package [Ora16] for Oracle DBMS offers linear algebra operations defined over UTL_NLA_ARRAY. UDFs provide a technical interface but do not define matrix operations over relations. No systematic approach to maintain contextual information is provided.

Luo et al. [LGG+17] extend SimSQL [CVP+13], a Hadoop-based relational system, with linear algebra functionality. RasDaMan [BDF+98, MB15] manages and processes image raster data. Both systems introduce *matrices* as ordered numeric-only attribute types. Although relations and matrices coexist, operations are defined over different objects. Linear operations are not defined over unordered objects and they do not support contextual information for individual cells of a matrix.

SciQL [ZKIN11, ZKM13] extends MonetDB [Mon17] with a new data type, `ARRAY`, as a first-class object. An array is stored as an object on its own. Arrays have a fixed schema: The last attribute stores the data values of a matrix, all other attributes are dimension attributes and must be numeric. Arrays come with a limited set of operations, such as addition, filtering, and aggregation, and they must be converted to relations to perform relational operations. The presence of contextual information and its inheritance are not addressed.

The MADlib library [HRS+12] for in-database analytics offers a broad range of linear and statistical operations, defined as either UDFs with C++ implementations or Eigen library calls. Matrix operations require a specific input format: Tables must have one attribute with a row id value and another array-valued attribute for matrix rows. Matrix operations return purely numeric results and cannot be nested.

Hutchison et al. [HHS17] propose LARA, an algebra with tuple-wise operations, attribute-wise operations, and tuple extensions. LARA defines linear and relational algebra operations using the same set of primitives. This is a good basis for inter-algebra optimizations that span linear and relational operations. LARA offers a strong theoretical basis, works out properties of the solution, and allows to store row and column descriptions during the operations. The maintenance of contextual information is not considered for operations that change the number of rows or columns.

LevelHeaded [ALOR18, ATOR16], an engine for relational and linear algebra operations, uses a special key-value structure: Each object has keys (dimension attributes) and annotations (value attributes). Dimension and value attributes are stored in a trie and a flat columnar buffer, respectively. Linear operations are available through an extended SQL syntax. Key values guarantee contextual information for rows. However, the trie key structure restricts relational operations:

For example, aggregations of keys and join predicates over non-key attributes (i.e., subselects in SQL) are not allowed.

SciDB [SBPR11] is a DBMS that is based on arrays. Matrices and relations are implemented as nested arrays. SciDB focuses on efficient array processing and performs linear algebra operations over arrays. SciDB supports element-wise operations and selected linear operations, such as SVD. The system also offers relational algebra operations on arrays but cannot compete with relational DBMSs such as MonetDB in terms of performance. A systematic approach to maintain contextual information is not considered.

Statistical packages, such as R [Pro18] and pandas [McK11], offer a broad range of linear and relational algebra operations over arrays. Each cell may be associated with descriptive information, but this information is not always inherited as part of operations (e.g., `usv`). No systematic solution for associating contextual information with numeric results is provided. The most important relational operations are supported, but even basic optimizations (e.g., join ordering) are missing.

The R package RIOT-DB [ZHY09] uses MySQL as a backend and translates linear computations to SQL. RIOT-DB addresses the main memory limitations of R, and the optimization of SQL statements yields inter-operation optimization. However, it is difficult (or sometimes impossible) to express linear algebra operations in SQL, and only a few simple operations, such as subtraction and multiplication, are discussed.

AIDA [DDMK18] integrates MonetDB and NumPy [Num18] and exploits the fact that both systems use C arrays as an internal data structure: To avoid copying NumPy data to MonetDB, AIDA passes pointers to arrays. Data copying is still needed to pass MonetDB results to NumPy since MonetDB does not guarantee that multiple columns are contiguous in memory, which is required by NumPy. AIDA offers a Python-like procedural language for relational and linear operations. Sequences of relational operations are evaluated lazily, which allows AIDA to combine and optimize sequences of relational operations. The optimization does not include linear algebra operations.

SystemML [GKP$^+$11] offers a set of linear algebra primitives that are expressed in a high-level, declarative language and are implemented on MapReduce. SystemML includes linear algebra optimizations that are similar to relational optimizations (e.g., selecting the order of execution of matrix multiplications). The system considers only linear algebra operations.

3.3 Preliminaries

This section presents notation for relations and matrices, and introduces the basic matrix algebra operations.

3.3.1 Relations

A relation r is a set of tuples r_i with schema $\mathscr{R}$. A schema, $\mathscr{R} = (A, B, \ldots)$, is a finite, ordered set of attribute names. A tuple $r_i \in r$ has a value from the appropriate domain for each attribute in the schema. We write $r_i.A$ to denote the value of attribute A in tuple r_i and $r.A$ to denote the set of all values $r_i.A$ in relation r. Ordered subsets of a schema, $\mathbf{U} \subseteq \mathscr{R}$, are typeset in bold. $|r|$ is the number of tuples in relation r.

Let r be a relation and $\mathbf{U} \subseteq \mathscr{R}$ be attributes that form a key of $\mathscr{R}$. We write $r^{\mathbf{U},k}$ to denote the k^{th} *tuple* of relation r sorted by the values of attributes $\mathbf{U}$ (in ascending order):

$$r_i = r^{\mathbf{U},k} \iff r_i \in r \wedge \\ |\{r_j \mid r_j \in r \wedge r_j.\mathbf{U} < r_i.\mathbf{U}\}| = k-1 \tag{3.1}$$

The *column cast* $\triangledown U$ creates an ordered set L from the sorted values of an attribute U that forms a key in relation r:

$$L = \triangledown U \iff |L| = |r| \wedge \\ \forall 1 \leq i \leq |r|(L[i] = r^{U,i}.U) \tag{3.2}$$

The column cast is used to generate a schema from a set of values. We use this for operations `tra`, `usv`, and `opd` (see Table 3.2). The column cast is applicable if the cardinality of a list of attributes $\mathbf{U}$ is one.

Example 9. Consider relation r in Figure 3.1. The third tuple of relation r sorted by the values of attribute V is $r^{(V),3} = (A, 30, 1)$, the column cast of O is $\triangledown O = (A, B, C)$, and the values of attribute W are $r.W = \{1, 5, 1\}$.

□

r

O	V	W
A	30	1
C	22	5
B	10	1

d

	1
1	D
2	B

e

	1	2
1	1	3
2	2	4

$d \square e$

	1	2	3
1	D	1	3
2	B	2	4

Figure 3.1: Relation r; matrices d, e, and $d \square e$

We use set notation and apply it to bags. Bags can be ordered or unordered. To emphasize the difference, parentheses are used for ordered bags (or lists), e.g., (3,2,3), and curly braces for unordered bags, e.g., {3,2,3}. When transitioning from unordered to ordered bags, the order is specified explicitly.

3.3.2 Matrices

An $n \times k$ matrix m is a two-dimensional array with n rows and k columns. $|m|$ is the number of rows, $\#m$ the number of columns. The element in the i^{th} row and the j^{th} column of matrix m is $m[i, j]$; the i^{th} row is $m[i, *]$; the j^{th} column is $m[*, j]$.

We consider the operations from the R Matrix Algebra [QR17]: element-wise multiplication (`EMU`), matrix multiplication (`MMU`), outer product (`OPD`), cross product (`CPD`), matrix addition (`ADD`), matrix subtraction (`SUB`), transpose (`TRA`), solve equation (`SOL`), inversion (`INV`), eigenvectors (`EVC`), eigenvalues (`EVL`), QR decomposition (`QQR`, `RQR`), SVD – single value decomposition (`DSV`, `USV`, `VSV`), determinant (`DET`), rank (`RNK`), and Choleski factorization (`CHF`). Note that QR and SVD return more than one matrix, therefore we split the operations: `QQR` and `RQR` return matrix Q and matrix R of the QR decomposition, respectively; `DSV`, `USV`, and `VSV` return vector D with the singular values, matrix U with the left singular vectors, and matrix V with the right singular vectors of SVD, respectively.

The *matrix concatenation* of matrices m and n with k rows each returns a matrix h with k rows. The i^{th} row of h is the concatenation of the i^{th} row of m and the i^{th} row of n.

$$\begin{aligned} h = m \square n \iff & |h| = |m| \wedge \\ & \forall 1 \leq i \leq |h| (h[i, *] = m[i, *] \circ n[i, *]) \end{aligned} \tag{3.3}$$

The *schema cast* $\Delta\mathbf{U}$ of attributes $\mathbf{U}$ creates a matrix m (with a single column) from the attribute names of $\mathbf{U}$:

$$m = \Delta\mathbf{U} \iff \#m = 1 \wedge |m| = |\mathbf{U}| \wedge \forall 1 \leq i \leq |\mathbf{U}|(m[i,1] = \mathbf{U}[i]) \tag{3.4}$$

Example 10. Consider attributes $\mathbf{U} = (D,B)$. Matrix d in Figure 3.1 is the result of the schema cast $d = \Delta\mathbf{U}$. The result of concatenating matrix d and matrix e is $d \square e$. Note that the row and column numbers (cells shaded in gray) in the matrix illustrations are not part of the matrix. □

Matrix operations are *shape restricted*, i.e., the number of result rows is equal to the number of *rows* of one of the input matrices (r), the number of *columns* of one of the input matrices (c), or *one* (1). The same holds for the number of result columns.

The dimensionality of result matrices defines the *shape type* of matrix operations. We write r_1 if the result dimensionality is equal to the number of rows in the first matrix, r_2 if the result dimensionality is equal to the number of rows in the second matrix, and r_* if the result dimensionality is equal to the number of rows in the first and second matrix (i.e., $\mathrm{r}_1 = \mathrm{r}_2$). The same notation holds for the number of columns. Table 3.1 summarizes the shape types of matrix operations.

Cardinalities	Shape type	Operations
$\|i_1 \times j_1\| \rightarrow \|i_1 \times i_1\|$	(r_1,r_1)	USV
$\|i_1 \times j_1\|, \|i_2 \times j_1\| \rightarrow \|i_1 \times i_2\|$	(r_1,r_2)	OPD
$\|i_1 \times i_1\| \rightarrow \|i_1 \times i_1\|$	(r_1,c_1)	INV, EVC, CHF
$\|i_1 \times j_1\| \rightarrow \|i_1 \times j_1\|$	(r_1,c_1)	QQR
$\|i_1 \times j_1\|, \|j_1 \times j_2\| \rightarrow \|i_1 \times j_2\|$	(r_1,c_2)	MMU
$\|i_1 \times i_1\| \rightarrow \|i_1 \times 1\|$	(r_1,1)	EVL
$\|i_1 \times j_1\| \rightarrow \|i_1 \times 1\|$	(r_1,1)	VSV
$\|i_1 \times j_1\| \rightarrow \|j_1 \times i_1\|$	(c_1,r_1)	TRA
$\|i_1 \times j_1\| \rightarrow \|j_1 \times j_1\|$	(c_1,c_1)	RQR , DSV
$\|i_1 \times j_1\|, \|i_1 \times j_2\| \rightarrow \|j_1 \times j_2\|$	(c_1,c_2)	CPD
$\|i_1 \times j_1\|, \|i_1 \times 1\| \rightarrow \|j_1 \times 1\|$	(c_1,c_2)	SOL
$\|i_1 \times j_1\|, \|i_1 \times j_1\| \rightarrow \|i_1 \times j_1\|$	(r_*,c_*)	EMU, ADD, SUB
$\|i_1 \times i_1\| \rightarrow \|1 \times 1\|$	(1,1)	DET
$\|i_1 \times j_1\| \rightarrow \|1 \times 1\|$	(1,1)	RNK

Table 3.1: Shape types of matrix operations

Example 11. Matrix multiplication has shape type (r_1,c_2), which states that the number of result rows is equal to the number of rows of the first argument matrix, and the number of columns is equal to the number of columns of the second argument matrix. Matrix addition has shape type (r_*,c_*), which states that the number of result rows is equal to the number of rows of the first matrix and the number of rows of the second matrix. □

All operations of the matrix algebra are *shape restricted*. This follows directly from the definitions of the matrix operations [GVL96]. The first column of Table 3.1 lists the relevant cardinalities from these definitions. We use shape restriction to determine the inheritance of contextual information. It has also been used in size propagation techniques [BKYJ19] for the purpose of cost-based optimization of chains of matrix operations.

3.4 The Relational Matrix Algebra

To seamlessly integrate matrix operations into the relational model, we extend the relational algebra to the *relational matrix algebra* (RMA). For each of the matrix operations we define a corresponding relational matrix operation in RMA: `emu`, `mmu`, `opd`, `cpd`, `add`, `sub`, `tra`, `sol`, `inv`, `evc`, `evl`, `qqr`, `rqr`, `dsv`, `usv`, `vsv`, `det`, `rnk`, `chf`. We use upper case for matrix operations (e.g., `TRA`) and lower case for RMA operations (e.g., `tra`). RMA includes both relational algebra and relational matrix operations. The new operations behave like regular operations with relations as input and output.

For each argument relation, r, of a relational matrix operation one parameter must be specified: The *order schema* $\mathbf{U} \subseteq \mathscr{R}$ imposes an order on the tuples for the purpose of the operation. The attributes of the order schema must form a key[3]. The attributes of relation r that are not part of the order schema $\overline{\mathbf{U}}$, i.e., $\overline{\mathbf{U}} = \mathscr{R} - \mathbf{U}$ form the *application schema*. The application schema identifies the attributes with the data to which the matrix operation is applied.

The order schema $\mathbf{U} \subseteq \mathscr{R}$ splits relation r into four non-overlapping areas: *Order schema* $\mathbf{U}$; *order part* $r.\mathbf{U}$; *application schema* $\overline{\mathbf{U}}$; and *application part* $r.\overline{\mathbf{U}}$. The parts of r that do not include matrix values, i.e., the order and application schemas ($\mathbf{U}$ and $\overline{\mathbf{U}}$) and the order part ($r.\mathbf{U}$), form the *contextual information* for application part $r.\overline{\mathbf{U}}$. Intuitively, the order schema and application schema provide context for columns while the order part provides context for rows.

[3] Attributes that neither belong to the order schema nor the application schema must be dropped explicitly with a projection (or added to the order schema, thus, forming a super key).

Example 12. Order schema $\mathbf{U} = (T)$ splits relation r in Figure 3.2 into four parts: Order schema $\mathbf{U} = (T)$, application schema $\overline{\mathbf{U}} = (H,W)$, order part $r.\mathbf{U} = r.(T) = \{5am, 8am, 7am, 6am\}$, and application part $r.\overline{\mathbf{U}} = r.(H,W) = \{(1,3), (8,5), (6,7), (1,4)\}$.

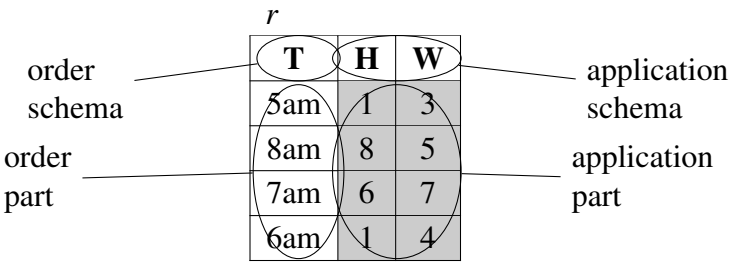


Figure 3.2: Structure of a relation instance

□

3.4.1 Matrix and Relation Constructors

Figure 3.3 summarizes our approach for the `inv` operation and example relation $r' = \sigma_{T>6am}(r)$: (1) two matrix constructors define matrices m and n that correspond to order and application part of r, respectively; (2) `INV` inverses matrix n resulting in matrix h; and (3) the relation constructor combines $m \square h$ and $\mathscr{R}$ into result relation v.

Definition 1. (*Matrix constructor*) Let r be a relation, $\mathbf{U}$ be an order schema. The matrix constructor $\mu_{\mathbf{U}}(r)$ returns a matrix that includes the values of $r.\mathbf{U}$ sorted by $\mathbf{U}$:

$$m = \mu_{\mathbf{U}}(r) \iff |m| = |r| \wedge \\ \forall 1 \leq i \leq |r|(m[i,*] = r^{\mathbf{U},i}.\mathbf{U})$$

We use the complement notation $\overline{\mu}_{\mathbf{U}}(r)$ to denote the matrix that includes the values of $r.\overline{\mathbf{U}}$ sorted by $\mathbf{U}$.

Example 13. Consider Figure 3.3 with relation $r' = \sigma_{T>6am}(r)$ and schema $\mathscr{R} = (T,H,W)$. The matrix constructor $\overline{\mu}_T(r')$ returns matrix n. □

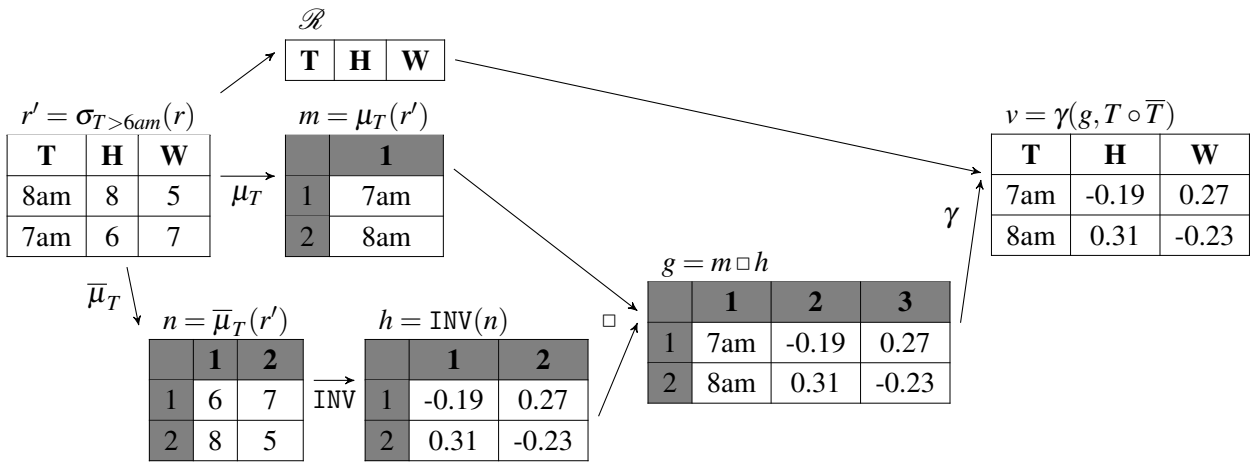


Figure 3.3: Structure of our solution for the inversion example, $v = \mathtt{inv}_T(\sigma_{T>6am}(r))$

Definition 2. (*Relation constructor*) Let m be a matrix with unique rows, and $\mathscr{R}$ be a relation schema with $\#m$ attributes. The relation constructor $\gamma(m, \mathscr{R})$ returns relation r with schema $\mathscr{R}$:

$$r = \gamma(m, \mathscr{R}) \iff |m| = |r| \wedge$$
$$\forall t (t \in r \iff \exists 1 \leq i \leq |m| (t = m[i, *]))$$

Example 14. In Figure 3.3, a relation constructor is applied to schema $\mathscr{R}$ and the concatenated matrices $m \square h$ to construct the result relation: $v = \gamma(m \square h, \mathscr{R})$. □

Matrix and relation constructors map between relations and matrices. We use constructors and matrices to define relational matrix operations and to analyze their properties. At the implementation level, constructors are very efficient since they split and combine lists of attribute names and *do not* access the data (cf. Section 3.7).

3.4.2 Relational Matrix Operations

Relational matrix operations offer the functionality of matrix operations in a relational context. The general form of a unary relational matrix operation is $\mathtt{op}_{\mathbf{U}}(r)$, where $\mathbf{U}$ is the order schema. A binary operation $\mathtt{op}_{\mathbf{U};\mathbf{V}}(r, s)$ has an additional order schema $\mathbf{V}$ for argument relation s.

The result of a relational matrix operation is a relation that consists of (a) the *base result* of the corresponding matrix operation, and (b) *contextual information*, appropriately morphed from the contextual information of the argument relations to reflect the semantics of the operation.

Definition 3. (*Base result*) Consider a unary relational matrix operation $\mathtt{op}_{\mathbf{U}}(r)$. The matrix that is the result of matrix operation $\mathtt{OP}(\overline{\mu}_{\mathbf{U}}(r))$ is the *base result* of $\mathtt{op}_{\mathbf{U}}(r)$. The base result for binary operations is defined analogously.

Shape type	Operations	Definition
(r$_1$,r$_1$)	usv	$\mathtt{op}_{\mathbf{U}}(r) = \gamma(\mu_{\mathbf{U}}(r) \,\square\, \mathtt{OP}(\overline{\mu}_{\mathbf{U}}(r)), \mathbf{U} \circ \triangledown\mathbf{U})$
(r$_1$,r$_2$)	opd	$\mathtt{op}_{\mathbf{U};\mathbf{V}}(r,s) = \gamma(\mu_{\mathbf{U}}(r) \,\square\, \mathtt{OP}(\overline{\mu}_{\mathbf{U}}(r), \overline{\mu}_{\mathbf{V}}(s)), \mathbf{U} \circ \triangledown\mathbf{V})$
(r$_1$,c$_1$)	inv, evc, chf, qqr	$\mathtt{op}_{\mathbf{U}}(r) = \gamma(\mu_{\mathbf{U}}(r) \,\square\, \mathtt{OP}(\overline{\mu}_{\mathbf{U}}(r)), \mathbf{U} \circ \overline{\mathbf{U}})$
(r$_1$,c$_2$)	mmu	$\mathtt{op}_{\mathbf{U};\mathbf{V}}(r,s) = \gamma(\mu_{\mathbf{U}}(r) \,\square\, \mathtt{OP}(\overline{\mu}_{\mathbf{U}}(r), \overline{\mu}_{\mathbf{V}}(s)), \mathbf{U} \circ \overline{\mathbf{V}})$
(r$_1$,1)	evl, vsv	$\mathtt{op}_{\mathbf{U}}(r) = \gamma(\mu_{\mathbf{U}}(r) \,\square\, \mathtt{OP}(\overline{\mu}_{\mathbf{U}}(r)), \mathbf{U} \circ (\mathtt{op}))$
(c$_1$,r$_1$)	tra	$\mathtt{op}^{C}_{\mathbf{U}}(r) = \gamma(\Delta\overline{\mathbf{U}} \,\square\, \mathtt{OP}(\overline{\mu}_{\mathbf{U}}(r)), (C) \circ \triangledown\mathbf{U})$
(c$_1$,c$_1$)	rqr, dsv	$\mathtt{op}^{C}_{\mathbf{U}}(r) = \gamma(\Delta\overline{\mathbf{U}} \,\square\, \mathtt{OP}(\overline{\mu}_{\mathbf{U}}(r)), (C) \circ \overline{\mathbf{U}})$
(c$_1$,c$_2$)	cpd, sol	$\mathtt{op}^{C}_{\mathbf{U};\mathbf{V}}(r,s) = \gamma(\Delta\overline{\mathbf{U}} \,\square\, \mathtt{OP}(\overline{\mu}_{\mathbf{U}}(r), \overline{\mu}_{\mathbf{V}}(s)), (C) \circ \overline{\mathbf{V}})$
(r$_*$,c$_*$)	emu, add, sub	$\mathtt{op}_{\mathbf{U};\mathbf{V}}(r,s) = \gamma(\mu_{\mathbf{U}}(r) \,\square\, \mu_{\mathbf{V}}(s) \,\square\, \mathtt{OP}(\overline{\mu}_{\mathbf{U}}(r), \overline{\mu}_{\mathbf{V}}(s)), \mathbf{U} \circ \mathbf{V} \circ \overline{\mathbf{U}})$
(1,1)	det, rnk	$\mathtt{op}^{C}_{\mathbf{U}}(r) = \gamma(r \circ \mathtt{OP}(\overline{\mu}_{\mathbf{U}}(r)), (C, \mathtt{op}))$

Table 3.2: Splitting and morphing relations and matrices

Example 15. Consider $\mathtt{inv}_T(\sigma_{T>6am}(r))$ in Fig. 3.3. The base result is matrix h, which results from $\mathtt{INV}(\overline{\mu}_T(\sigma_{T>6am}(r)))$. □

Table 3.2 defines the details of how contextual information is maintained in relational matrix operations. All definitions follow the structure illustrated in Figure 3.3. A result relation is composed from order parts, base result, and schemas with the help of a relation constructor. For example, inv is defined according to its shape type in Table 3.2: $\mathtt{inv}_{\mathbf{U}}(r) = \gamma(\mu_{\mathbf{U}}(r) \,\square\, \mathtt{INV}(\overline{\mu}_{\mathbf{U}}(r)), \mathbf{U} \circ \overline{\mathbf{U}})$, where $\mu_{\mathbf{U}}(r)$ are the rows of the order part, $\mathtt{INV}(\overline{\mu}_{\mathbf{U}}(r))$ is the base result, and $\mathbf{U} \circ \overline{\mathbf{U}}$ is the result schema.

Operations that have a different number of rows than any of the input relations add new attribute C to the result relation. This attribute C is for contextual information (cf. Example 16): Its values are either the attribute names of the application schema of an input relation or the operation name. The operations add, sub, emu require union compatible application schemas and non-overlapping order schemas. Operations usv, opd, and tra construct the application schema of the result from the order schema of an input relation. Therefore the cardinality of the order schemas $\mathbf{U}$ of tra and usv, and $\mathbf{V}$ of opd must be one.

Example 16. Consider Figures 3.2 and 3.4. Figure 3.4a illustrates the result of $\texttt{qqr}_T(r)$. The values of T define the ordering of tuples for this operation. The values of H and W are the values of matrix Q computed as part of the QR decomposition. Figure 3.4b illustrates the result of $\texttt{tra}^C_T(r)$. The column cast $\triangledown T$ of ordering attribute T provides names for the attributes in the transposed relation. The result relation has new attribute C whose values are the names of the attributes in the application schema of r. Note that all result relations come with sufficient contextual information for each value. For example, relation r in Figure 3.2 records that Humidity (H) was 1 at 6am, which is also recorded in the transposed relation in Figure 3.4b.

$\texttt{qqr}_T(r)$

T	H	W
5am	0.1	0.5
6am	0.8	-0.4
7am	0.6	0.4
8am	0.1	0.7

(a) QR decomposition

$\texttt{tra}^C_T(r)$

C	5am	6am	7am	8am
H	1	1	6	8
W	3	4	7	5

(b) Transpose

Figure 3.4: Examples of relational matrix operations

□

3.5 RMA in Action

This section gives an application example with a mixed workload that combines relational and linear algebra operations. It maintains all data in regular relations and illustrates the importance of maintaining contextual information.

Consider relations u, f, and r in Figure 3.5. Relation u records name, state, and year of birth of users; relation f records title, release year, and director of films; relation r records user ratings for films. Tuple u_1 states that user Ann lives in California and was born in 1980; tuple f_1 states that film Heat was directed by Lee and was released in 1995; tuple r_1 states that Ann's ratings for Balto, Heat, and Net are, respectively, 2.0, 1.5, and 0.5.

The task is to determine how similar each of Lee's films is to any other film, based on the ratings from California users. The covariance [JW07] is used to compute this similarity. In addition, we need relational algebra operations (e.g., selection σ, aggregation ϑ, rename ρ, and join $\bowtie$) to retrieve selected ratings and films, aggregate ratings, and combine information from different

u (user)

	User	State	YoB
u_1	Ann	CA	1980
u_2	Tom	FL	1965
u_3	Jan	CA	1970

f (film)

	Title	RelY	Director
f_1	Heat	1995	Lee
f_2	Balto	1995	Lee
f_3	Net	1995	Smith

r (rating)

	User	Balto	Heat	Net
r_1	Ann	2.0	1.5	0.5
r_2	Tom	0.0	0.0	1.5
r_3	Jan	1.0	4.0	1.0

Figure 3.5: Example database

tables. The key observation is that a mixture of matrix and relational operations is required to determine the similarities of the ratings.

The solution in Figure 3.6 includes three key steps: Data preparation ($w1$), covariance computation ($w2$-$w7$), and retrieving Lee's films together with all similarities ($w8$). Note the seamless integration of linear and relational algebra. The entire process frequently switches between linear and relational operations.

$$
\begin{aligned}
w1 &= \pi_{U,B,H,N}(\sigma_{S=\text{'CA'}}(u \bowtie r)) \\
w2 &= \vartheta_{AVG(B),AVG(H),AVG(N)}(w1) \\
w3 &= \pi_{U,B,H,N}(\mathtt{sub}_{U;V}(w1, \rho_V(\pi_U(w1)) \times w2)) \\
w4 &= \mathtt{tra}_U^T(w3) \\
w5 &= \mathtt{mmu}_{C;U}(w4, w3) \\
w6 &= w5 \times \rho_M(\vartheta_{COUNT(*)}(w1)) \\
w7 &= \pi_{C,B/(M-1),H/(M-1),N/(M-1)}(w6) \\
w8 &= \pi_{T,B,H,N}(\sigma_{D=\text{'Lee'}}(w7 \bowtie_{C=T} f))
\end{aligned}
$$

Figure 3.6: Computing the similarity of the ratings

In the following we discuss the algebra expressions in Figure 3.6. First, we join u and r to select ratings from California users ($w1$). Next, we compute the covariance using its standard definition [JW07]: $cov(X,Y) = \frac{1}{n-1}[(X - E[X]) * (Y - E[Y])^\mathsf{T}]$. The expectation of an attribute, e.g., $E(H) = \vartheta_{AVG(H)}(...)$, is computed via aggregation ($w2$). *Relational matrix operations*, `sub`, `tra` and `mmu`, are used to subtract ($X - E[X]$), transpose ($^\mathsf{T}$), and multiply ($*$) relations

$(w3, w4, w5)$. Next, we compute the unbiased covarinace $(w6, w7)$. Finally, we join $w7$ and f to select Lee's films.

Figure 3.7 illustrates relations $w3$, $w4$, and $w8$. Consider transpose $\mathtt{tra}^T_U(w3)$ with order schema U and application schema $\overline{U} = (B, H, N)$. The result of this operation is a relation $w4$ with schema (*C*,*Ann*,*Jan*). The values of attribute *C* are the attribute names in the application schema of $w3$. Note that each operation preserves schema and ordering information as the crucial parts of contextual information. This makes it possible to interpret the tuples in result relation $w8$. For example, tuple z_1 states that Lee's film Balto has the smallest covariance to film Net.

w3

U	**B**	**H**	**N**
Ann	-1.25	0.5	0.25
Jan	1.25	-0.5	0.25

w4

C	**Ann**	**Jan**
B	-1.25	1.25
H	-0.5	0.5
N	-0.25	0.25

w8

	T	**B**	**H**	**N**
z_1	B	1.56	-0.62	-2.5
z_2	H	-0.62	0.25	1

Figure 3.7: Steps during the computation

3.6 Properties of RMA

This section defines two crucial requirements for relational matrix operations. *Matrix consistency* guarantees that the result of a relational matrix operation can be reduced to the result of the corresponding matrix operation. *Origins* guarantee that each result relation includes sufficient inherited contextual information to relate argument and result relation. We prove that each relational matrix operation is matrix consistent and returns a relation with origins.

3.6.1 Matrix Consistency

Matrix consistency ensures that the result relation includes all cell values that are present in the base result and the order of rows in the base result can be derived from contextual information in the result relation. First, we define *reducibility* to transition from relations to matrices.

Definition 4. (*Reducible*) Let r be a relation, $\mathbf{U}$ be an order schema. Relation r is *reducible* to matrix m iff m can be constructed from the attribute values of $\overline{\mathbf{U}}$ in relation r sorted by $\mathbf{U}$:

$$r \rightarrow_{\mathbf{U}} m \quad \Longleftrightarrow \quad \overline{\mu}_{\mathbf{U}}(r) = m$$

Example 17. Consider Fig. 3.3 with relation $r' = \sigma_{T>6am}(r)$, matrix n, and order schema T. From Example 13 we have $\overline{\mu}_T(r') = n$. Relation r' is reducible to matrix n since n can be constructed from the values of H and W in the argument relation sorted by T, i.e., $r' \rightarrow_T n$. □

Definition 5. (*Matrix consistency*) Consider a unary matrix operation $\texttt{OP}(m)$. The corresponding relational matrix operation op is *matrix consistent* iff for all relations r that are reducible to matrix m, the result relation $\texttt{op}_{\mathbf{U}}(r)$ is reducible to $\texttt{OP}(m)$:

$$\forall r, m, \mathbf{U}(r \rightarrow_{\mathbf{U}} m \implies \exists \mathbf{U}'(\texttt{op}_{\mathbf{U}}(r) \rightarrow_{\mathbf{U}'} \texttt{OP}(m)))$$

A binary relational matrix operation is matrix consistent if its result is reducible to the result of the corresponding binary matrix operation.

Example 18. Consider Figures 3.2 and 3.8 with relation r, matrix g, matrix $\texttt{RQR}(g)$ and relation $\texttt{rqr}^C_T(r)$.

- $r \rightarrow_T g$: Relation r is reducible to matrix g
- $\texttt{rqr}^C_T(r) \rightarrow_C \texttt{RQR}(g)$: relation $\texttt{rqr}^C_T(r)$ is reducible to matrix $\texttt{RQR}(g)$

g

	1	**2**
1	1	3
2	1	4
3	6	7
4	8	5

$\texttt{RQR}(g)$

	1	**2**
1	-10.1	-8.8
2	0.0	-4.6

$\texttt{rqr}^C_T(r)$

C	**H**	**W**
H	-10.1	-8.8
W	0.0	-4.6

Figure 3.8: Example of matrix consistency

□

3.6.2 Origins of Result Relations

The result of a relational matrix operation is a relation that, in addition to the base result, includes a *row origin* and a *column origin*. Origins (1) uniquely define the relative positioning of result values, (2) give a meaning to values with respect to the applied operation, and (3) establish a connection between argument relations of an operation and its result relation.

Example 19. Consider inversion and result relation v in Figure 3.3. Values 7*am* and 8*am* show that (1) value -0.19 precedes value 0.31 because 7*am* precedes 8*am*; (2) -0.19 is the inversion value for humidity and for time 7*am*; (3) value -0.19 in relation v is connected to value 6 in the argument relation since they have the same origins (7*am* and H). □

Origins are either inherited order schemas, application schemas from argument relations, or constants. The shape type of an operation determines the cardinality of inherited contextual information. The indices in the shape type specify the input relation, from which an origin is inherited. For example, if the first element of the shape type is c_1, the row origin is the schema cast of the application schema of the first argument relation. Note that indices $_*$ and $_2$ are only possible for binary operations.

Definition 6. (*Origins*) Consider a unary, $v = \mathtt{op}_{\mathbf{U}}(r)$, or binary, $v = \mathtt{op}_{\mathbf{U};\mathbf{V}}(r,s)$, matrix consistent operation with shape type (x,y), base result m, and attribute list $\mathbf{U}'$ such that $v \rightarrow_{\mathbf{U}'} m$. Consider Table 3.3. $v.\mathbf{U}'$ is a *row origin* iff $v.\mathbf{U}'$ is equal to *ro* for the given shape type x. $\overline{\mathbf{U}'}$ is a *column origin* iff $\overline{\mathbf{U}'}$ is equal to *co* for the given shape type y.

x	*ro*
r_1	$r.\mathbf{U}$
r_2	$s.\mathbf{V}$
c_1	$\Delta\overline{\mathbf{U}}$
c_2	$\Delta\overline{\mathbf{V}}$
r_*	$(r.\mathbf{U}, s.\mathbf{V})$
c_*	$(\Delta\overline{\mathbf{U}}, \Delta\overline{\mathbf{V}})$
1	$'r'$

y	*co*
r_1	$\nabla\mathbf{U}$
r_2	$\nabla\mathbf{V}$
c_1	$\overline{\mathbf{U}}$
c_2	$\overline{\mathbf{V}}$
r_*	$\nabla\mathbf{U}$
c_*	$\overline{\mathbf{U}}$
1	$'op'$

Table 3.3: Definition of origins for shape type (x,y)

Example 20. Figure 3.9 illustrates relation r and the origins for operations

- $\mathtt{rnk}_H^C(\pi_{H,W}(r))$ with shape type (1,1)
- $\mathtt{usv}_T(r)$ with shape type (r_1,r_1)
- $\mathtt{qqr}_{W,T}(r)$ with shape type (r_1,c_1)

Column origins (*co*) are marked by rectangles (all values inside a rectangle form together the column origin for the relation). *Row origins* (*ro*) are marked by ellipses.

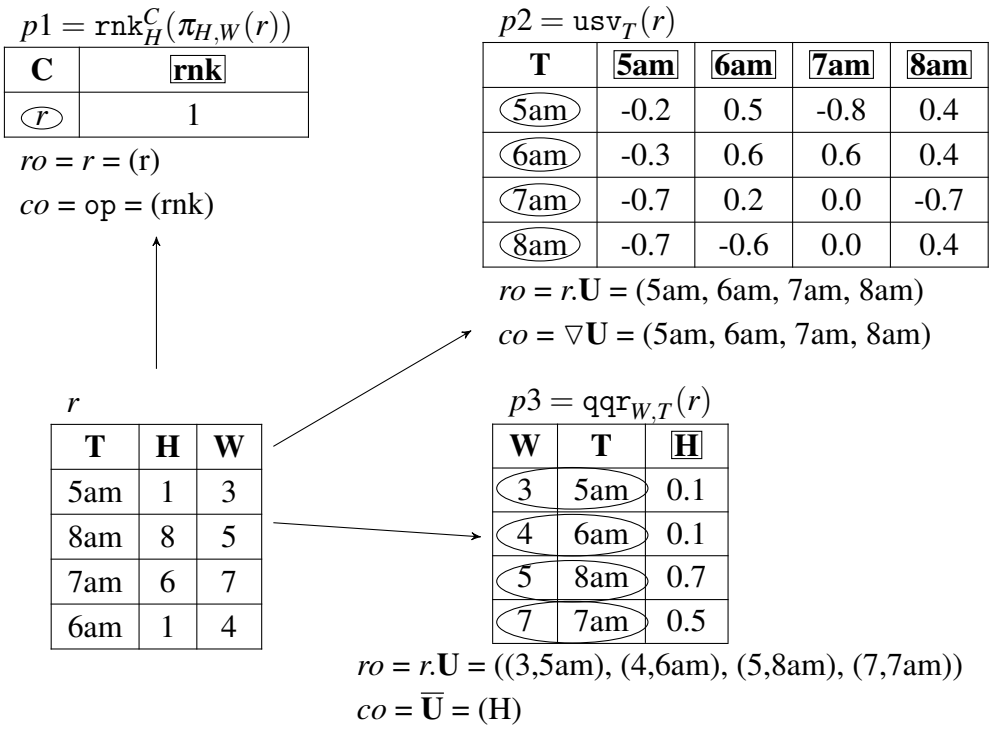


Figure 3.9: Examples of origins

For $p2 = \texttt{usv}_T(r)$, we have $\mathbf{U} = (T)$, $\overline{\mathbf{U}} = (H,W)$, $\mathbf{U}' = (T)$, and $\overline{\mathbf{U}'}$ = (5am, 6am, 7am, 8am). The shape type of usv is (r_1, r_1) (see Table 3.2) this makes $p2.T = r.T$ a row origin and $\triangledown T$ = (5am, 6am, 7am, 8am) a column origin. □

3.6.3 Correctness

Theorem 1. *All relational matrix operations return matrix consistent relations with a row and column origin.*

Proof. First, we prove that relational matrix operations are matrix consistent. Second, we show that inherited contextual information in the result corresponds to Definition 6.

(1) Consider a unary operation with shape type (x,y). We start with the definition of matrix consistency (Definition 5), $\mathbf{U}'$ equal to $\mathbf{U}$ (x = r_1) or C (x = c_1), and $\overline{\mathbf{U}'}$ equal to $\overline{\mathbf{U}}$ (y = c_1) or $\triangledown\mathbf{U}$ (y = r_1). We instantiate the implication with the definition of the operation in Table 3.2. Then, we simplify the right hand side of the implication, substituting it with the equality in Definition 4. Next, we expand the equality with the definitions of relational and matrix constructors

(Definitions 1 and 2). A series of simplifications yields the equality of the right and left hand side.

(2) Consider operation $v = \texttt{qqr}_{\mathbf{U}}(r)$ with shape type (r_1, c_1). Matrix consistency of qqr was shown in the first part of the proof. The shape type of qqr is (r_1,c_1) and we get $\overline{\mathbf{U}} = \overline{\mathbf{U}'}$, and $\mathbf{U} = \mathbf{U}'$. $v.\mathbf{U}$ is the row origin because x=r_1 and $\overline{\mathbf{U}}$ is the column origin because y=c_1 (see Table 3.3).

The same reasoning applies to the other types of operations in Table 3.2. Thus, each relational matrix operation returns a result relation with a row and a column origin. □

The following example uses a sequence of tra operations to illustrate the importance of origins. A result relation with origins inherits sufficient contextual information, such that each value can be interpreted. Origins also carry sufficient information about the order of rows, such that in sequences of relational matrix operations no ordering information is lost between operations.

Example 21. Consider a relation r that is reducible to matrix n. Figure 3.10 shows the matrix expression $\texttt{TRA}(\texttt{TRA}(n))$ and the respective relational matrix expression $\texttt{tra}_C^C(\texttt{tra}_T^C(r))$. Operation $\texttt{tra}_T^C(r)$ returns relation $r1$, which in addition to the application schema (attributes 5*am*, 6*am*, 7*am*, 8*am*) also includes attribute C, which is preserved together with the application schema.

□

3.7 Implementation

We discuss the integration of our solution into MonetDB. The implementation of relational matrix operations includes the processing of contextual information and the computation of the base result. Contextual information is handled inside MonetDB, while the computation of the base result can be done in MonetDB or delegated to external libraries (e.g., MKL). The integration of each relational matrix operation requires extensions throughout the system, but does not change the query processing pipeline and no new data structures are introduced. To extend MonetDB with addition, QR decomposition, linear regression, and the transformation of numerical data to the MKL format we touch 20 (out of 4500) files and add 2500 lines of code.

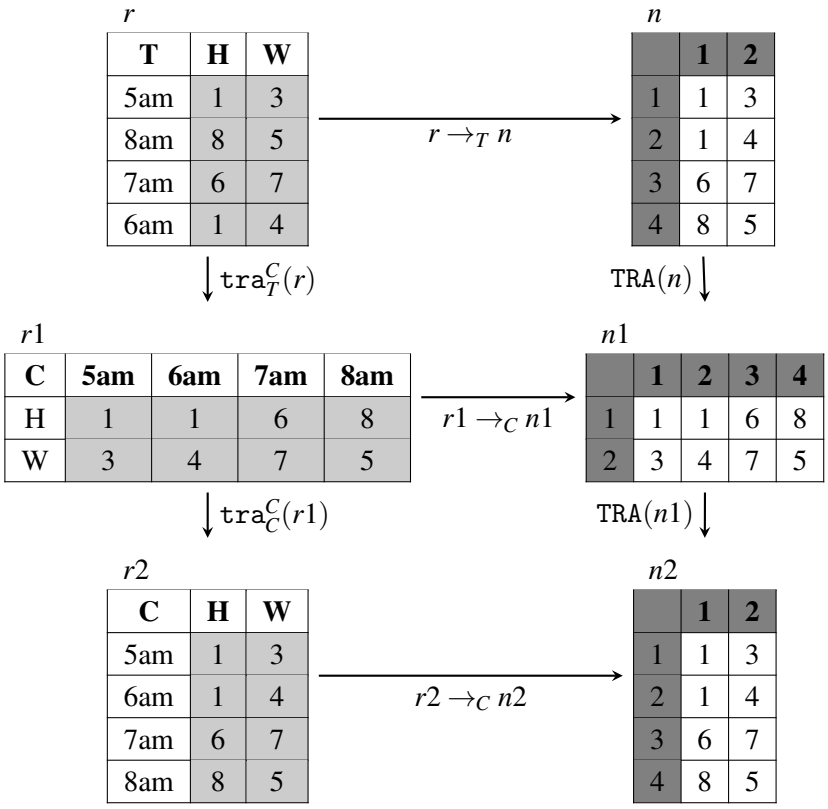


Figure 3.10: Origins and matrix consistency

3.7.1 MonetDB

MonetDB stores each column of a table as a binary association table (BAT). A BAT is a table with two columns: Head and tail. The head is a column with object identifiers (OID), while the tail is a column with attribute values. All attribute values of a tuple in a relation have the same OID value. Thus, a tuple can be constructed by concatenating all tail values with the same OID. MonetDB operations manipulate BATs and relational operations are represented and executed as sequences of BAT operations. Example BAT operations are $B_1 * B_2$, B_1/B_2, and $B_1 - B_2$ for element-wise multiplication, division, and subtraction, and $sum(B)$ to sum the values in BAT B.

Example 22. MonetDB stores relation $\sigma_{T>6am}(r)$ from Figure 3.3 in three BATs as shown in Figure 3.14.

□

T

OID	Val
0	8am
1	7am

H

OID	Val
0	8
1	6

W

OID	Val
0	5
1	7

Figure 3.11: BAT representation of $\sigma_{T>6am}(r)$

One important BAT operation is *leftfetchjoin* ($\downarrow$), which returns a BAT with OIDs sorted according to the order of OIDs of another BAT from the same relation. For instance, $X \downarrow Y$ returns BAT X, whose OIDs have the same order as OIDs of BAT Y. $X \downarrow X$ denotes X sorted by its own values.

The internal representation of a sequence of BAT operations is called MAL (MonetDB Assembly Language) plan, which consists of MAL instructions. For example, the MAL instruction $batcalc.*(B_1, B_2)$ corresponds to the BAT multiplication $B_1 * B_2$.

3.7.2 RMA Integration

As a first step, we have extended the SQL parser to make the relational matrix operations available in the from clause of SQL [DAB20a]. The syntax `(r BY U)` specifies ordering for argument relation r. As an example, consider relations r and s and ordering attributes **U** and **V**. The unary operation $\texttt{inv}_{\mathbf{U}}$ and the binary operation $\texttt{mmu}_{\mathbf{U};\mathbf{V}}$ are expressed as shown in Figure 3.12.

```
1  SELECT * FROM INV(r BY U);
2  SELECT * FROM MMU(r BY U, s BY V);
```

Figure 3.12: `inv` and `mmu` syntax

These basic constructs can be composed to more complex expressions. For instance, folding $w5$, $w6$ and $w7$ from Figure 3.6 yields the RMA expression:

$$\pi_{C,B/(M-1),H/(M-1),N/(M-1)}(\\ \texttt{mmu}_{C;U}(w4, w3) \times \rho_M(\vartheta_{COUNT(*)}(w1)))$$

Figure 3.13 illustrates the SQL translation of this expression.

Algorithm 3 processes a node that represents a unary relational matrix operation $\texttt{op}_{\mathbf{U}}(r)$ and translates it to a list of BAT expressions. In lines 2 - 7, the BATs of relation r are split, sorted, and morphed to get BATs X with row origins and BATs Y with the application part. *Splitting* (lines 2

```
1  SELECT C, B/(M-1), H/(M-1), N/(M-1)
2  FROM MMU( w4 BY C, w3 BY U ) AS w5
3  CROSS JOIN
4  ( SELECT COUNT(*) AS M FROM w1 ) AS t;
```

Figure 3.13: SQL translation

Algorithm 3: UnaryRMA(op, **U**, r)

1 $B \leftarrow BATs(r)$; $Y \leftarrow \{\}$;
2 $D \leftarrow \{b | b \in B, b$ *is in order schema* $\mathbf{U}\}$;
3 $G \leftarrow sort(D)$;
4 **for** $b \in B \setminus D$ **do** $Y \leftarrow Y \cup b \downarrow G$;
5 **if** *ShapeType*(op) $\in$ *{(r,r), (r,c), (r,1)}* **then** $X \leftarrow G$;
6 **else if** *ShapeType*(op) $\in$ *{(c,r), (c,c)}* **then** $X \leftarrow newBAT(Y)$;
7 **else** $X \leftarrow newBAT(r)$;
8 $F \leftarrow eval(\mathtt{op}, Y)$;
9 **return** $Concat(X, F)$;

and 4) divides a relation into two parts: The application part, on which the matrix operations are performed, and the contextual information, which gives a meaning to the application part. BATs B are split into application part and order part according to **U**. *Sorting* (lines 3 and 4) determines the order of the tuples for a specific matrix operation. Order schema **U** is used to sort the BATs: BATs in **U** are sorted according to their values while the other BATs in B are sorted according to the OIDs of the BATs in **U**. The order is established for each operation based on the contextual values in the relation. *Morphing* (lines 5-7) morphs the contextual information so that it can be added to the base result. Finally, the matrix operation is applied to Y (line 8). *Merging* (line 9) combines the result of the matrix operation with relevant contextual information and constructs the result relation with row and column origins. Merging and splitting are efficient operations that work at the schema level and do not access the data.

Example 23. Figure 3.14 illustrates Algorithm 3 for $v = \mathtt{inv}_T(\sigma_{T>6am}(r))$. *Splitting*: Input list $B = (T, H, W)$ is split into order list $D = (T)$ and application list $B \setminus D = (H, W)$. *Sorting*: BAT T is sorted, producing G. Then, (H, W) are sorted according to G returning $(H \downarrow T, W \downarrow T)$. *Morphing*: Since inv is of shape type (r_1,c_1), row contextual information is the order part: $X = T \downarrow T$. *Merging*: BATs X are concatenated with BATs F to form the result. □

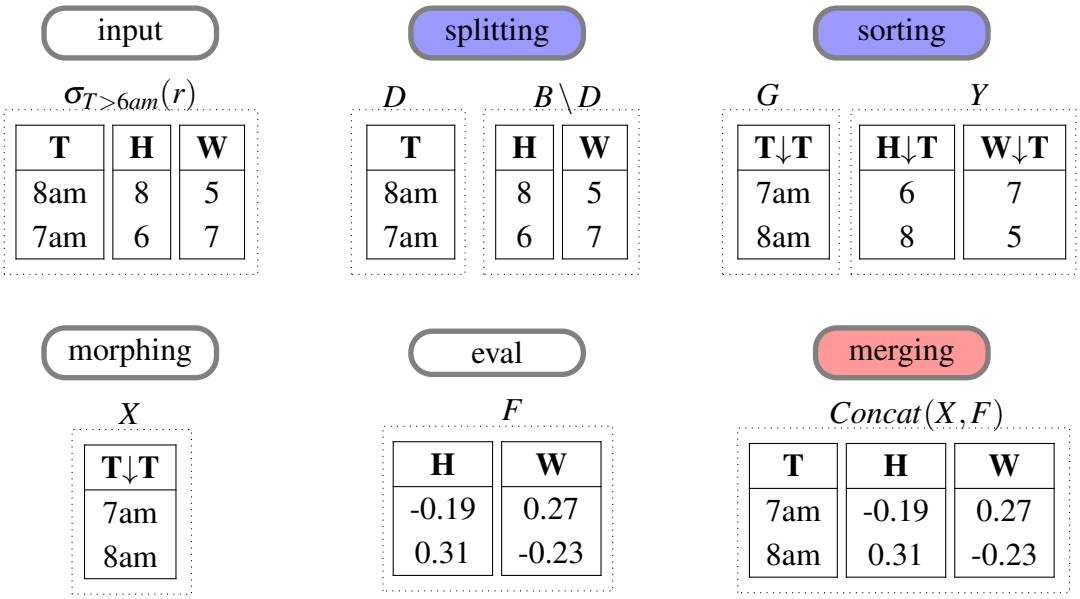


Figure 3.14: Splitting, sorting, morphing, merging for query $v = \texttt{inv}_T(\sigma_{T>6am}(r))$

3.7.3 Computing the Base Result

Line 8 of Algorithm 3 calls the procedure that computes the matrix operation. The computation can be done either in the kernel of MonetDB or by calling an external library (e.g., MKL [Int20]). Calling an external library requires copying data from BATs to the external format and copying the result back. The query optimizer decides about external library calls based on the complexity of the operation, the amount of data to be copied, and the relative performance of the matrix operation in MonetDB compared to the external library.

The no-copy implementation of matrix operations in the kernel of MonetDB is performed over BATs directly. Essentially, standard algorithms must be reduced to BAT operations. The process of reducing is highly dependent on the operation. The goal is to design algorithms that access entire columns and minimize accesses to single elements of BATs. To achieve this standard value-based algorithms must be transformed to vectorized BAT operations.

Algorithm 4 illustrates the reduction for the Gauss Jordan elimination method for the `INV` computation. The algorithm takes a list of BATs $B = (B_1, B_2, .., B_n)$ and returns the inversion as a list of BATs BR of the same size. Function $IDmatrix(n)$ creates a list of BATs that represents the identity matrix of size $n \times n$. The selection operation $sel(B,i)$ returns the i^{th} value in B. With the exception of the *sel* operation, all operations are standard MonetDB BAT operations that are also

Algorithm 4: INV(B)

```
1  n ← B.length;
2  BR ← IDmatrix(n);
3  for i = 1 to n do
4  |   v1 ← sel(B_i, i);
5  |   B_i ← B_i / v1;
6  |   BR_i ← BR_i / v1;
7  |   for j = 1 to n do
8  |   |   if i ≠ j then
9  |   |   |   v2 ← sel(B_j, i);
10 |   |   |   B_j ← B_j − B_i * v2;
11 |   |   |   BR_j ← BR_j − BR_i * v2;
12 return BR;
```

used for relational queries. For example, the operation on $B_i \leftarrow B_i/v_1$; divides each element of a BAT with a scalar value.

Example 24. Consider relation $r1$ with schema $r1(V1, A1, A2)$, where $V1$ is the order schema and $A1, A2$ is the application schema. The MAL plan in Listing 3.1 illustrates how Algorithm 4, applied to instance $r1$ with two tuples, is evaluated by MonetDB.

Listing 3.1: MAL plan for `inv`

```
1  sql>explain select * from inv (r1 by v1);
2  mal
3  function user.s6_1():void;
4      X_1:void := querylog.define("select * from inv (r1 by v1);":str,
5          "default_pipe":str, 80:int);
6      barrier X_159:bit := language.dataflow();
7      X_43:bat[:dbl] := bat.new(nil:dbl);
8      X_4:int := sql.mvc();
9      C_5:bat[:oid] := sql.tid(X_4:int, "sys":str, "r1":str);
10     X_8:bat[:str] := sql.bind(X_4:int, "sys":str, "r1":str, "v1":str, 0:int);
11     X_17:bat[:str] := algebra.projection(C_5:bat[:oid], X_8:bat[:str]);
12     (X_32:bat[:str], X_33:bat[:oid], X_34:bat[:oid]) :=
13         algebra.sort(X_17:bat[:str], false:bit, false:bit);
14     X_25:bat[:dbl] := sql.bind(X_4:int, "sys":str, "r1":str, "a2":str, 0:int);
15     X_40:bat[:dbl] := algebra.projectionpath(X_33:bat[:oid],
16         C_5:bat[:oid], X_25:bat[:dbl]);
17     X_18:bat[:dbl] := sql.bind(X_4:int, "sys":str, "r1":str, "a1":str, 0:int);
18     X_39:bat[:dbl] := algebra.projectionpath(X_33:bat[:oid],
19         C_5:bat[:oid], X_18:bat[:dbl]);
20     X_37:bat[:str] := algebra.projection(X_33:bat[:oid], X_17:bat[:str]);
21     X_45:bat[:dbl] := bat.append(X_43:bat[:dbl], 1:dbl, true:bit);
22     X_49:bat[:dbl] := bat.append(X_45:bat[:dbl], 0:dbl, true:bit);
23     X_51:bat[:dbl] := bat.new(nil:dbl);
```

```
24      X_52:bat[:dbl] := bat.append(X_51:bat[:dbl], 0:dbl, true:bit);
25      X_54:bat[:dbl] := bat.append(X_52:bat[:dbl], 1:dbl, true:bit);
26      X_82:bat[:str] := bat.new(nil:str);
27      X_88:bat[:int] := bat.new(nil:int);
28      X_86:bat[:int] := bat.new(nil:int);
29      X_85:bat[:str] := bat.new(nil:str);
30      X_84:bat[:str] := bat.new(nil:str);
31      X_58:dbl := batcalc.sel(X_39:bat[:dbl], 0:int);
32      X_60:bat[:dbl] := batcalc./(X_49:bat[:dbl], X_58:dbl);
33      X_59:bat[:dbl] := batcalc./(X_39:bat[:dbl], X_58:dbl);
34      X_76:dbl := batcalc.sel(X_59:bat[:dbl], 1:int);
35      X_63:dbl := batcalc.sel(X_40:bat[:dbl], 0:int);
36      X_66:bat[:dbl] := batcalc.*(X_63:dbl, X_60:bat[:dbl]);
37      X_67:bat[:dbl] := batcalc.-(X_54:bat[:dbl], X_66:bat[:dbl]);
38      X_64:bat[:dbl] := batcalc.*(X_63:dbl, X_59:bat[:dbl]);
39      X_65:bat[:dbl] := batcalc.-(X_40:bat[:dbl], X_64:bat[:dbl]);
40      X_71:dbl := batcalc.sel(X_65:bat[:dbl], 1:int);
41      X_73:bat[:dbl] := batcalc./(X_67:bat[:dbl], X_71:dbl);
42      X_79:bat[:dbl] := batcalc.*(X_76:dbl, X_73:bat[:dbl]);
43      X_80:bat[:dbl] := batcalc.-(X_60:bat[:dbl], X_79:bat[:dbl]);
44      X_89:bat[:str] := bat.append(X_82:bat[:str], "sys.r1":str);
45      X_91:bat[:str] := bat.append(X_84:bat[:str], "v1":str);
46      X_93:bat[:str] := bat.append(X_85:bat[:str], "clob":str);
47      X_95:bat[:int] := bat.append(X_86:bat[:int], 0:int);
48      X_97:bat[:int] := bat.append(X_88:bat[:int], 0:int);
49      X_98:bat[:str] := bat.append(X_89:bat[:str], ".L2":str);
50      X_100:bat[:str] := bat.append(X_91:bat[:str], "a1":str);
51      X_102:bat[:str] := bat.append(X_93:bat[:str], "double":str);
52      X_104:bat[:int] := bat.append(X_95:bat[:int], 53:int);
53      X_106:bat[:int] := bat.append(X_97:bat[:int], 0:int);
54      X_107:bat[:str] := bat.append(X_98:bat[:str], ".L2":str);
55      X_108:bat[:str] := bat.append(X_100:bat[:str], "a2":str);
56      X_110:bat[:str] := bat.append(X_102:bat[:str], "double":str);
57      X_111:bat[:int] := bat.append(X_104:bat[:int], 53:int);
58      X_112:bat[:int] := bat.append(X_106:bat[:int], 0:int);
59      language.pass(C_5:bat[:oid]);
60      language.pass(X_33:bat[:oid]);
61      language.pass(X_17:bat[:str]);
62      language.pass(X_39:bat[:dbl]);
63      language.pass(X_59:bat[:dbl]);
64      language.pass(X_40:bat[:dbl]);
65      language.pass(X_60:bat[:dbl]);
66      exit X_159:bit;
67      sql.resultSet(X_107:bat[:str], X_108:bat[:str], X_110:bat[:str],
68                    X_111:bat[:int], X_112:bat[:int], X_37:bat[:str],
69                    X_80:bat[:dbl], X_73:bat[:dbl]);
70  end user.s6_1;
```

In lines 9-20 the input BATs are declared and sorted. The creation of BATs for a 2×2 identity matrix, which corresponds to function $IDMatrix(2)$ from Algorithm 4, is done in lines 21-25. Lines 31-43 correspond to the outer for-loop from Algorithm 4. For example, the selection of values is shown in lines 31, 34, 35, and 40. The rest of the statements are responsible for the meta information of BATs, such as their names.

Example 25. Consider relations r and s with schemas $r(U1, A1, A2)$ and $s(V1, V2, B1, B2)$, where $U1$ and $V1$ are strings and $A1, A2, B1, B2$ are doubles.

Listing 3.2 illustrates the MAL plan that corresponds to the following RMA query:

```
1   SELECT * FROM ADD (r BY U1, s BY V1, V2);
```

Listing 3.2: MAL plan for `add`

```
1  sql>explain select * from add (r by u1, s by v1, v2);
2  mal
3  function user.s4_1():void;
4      X_1:void := querylog.define("explain select * from add (r by u1,
5                   s by v1, v2);":str, "default_pipe":str, 86:int);
6      barrier X_175:bit := language.dataflow();
7      X_85:bat[:str] := bat.new(nil:str);
8      X_91:bat[:int] := bat.new(nil:int);
9      X_89:bat[:int] := bat.new(nil:int);
10     X_88:bat[:str] := bat.new(nil:str);
11     X_87:bat[:str] := bat.new(nil:str);
12     X_4:int := sql.mvc();
13     C_5:bat[:oid] := sql.tid(X_4:int, "sys":str, "r":str);
14     X_8:bat[:str] := sql.bind(X_4:int, "sys":str, "r":str, "u1":str, 0:int);
15     X_17:bat[:str] := algebra.projection(C_5:bat[:oid], X_8:bat[:str]);
16     (X_62:bat[:str], X_63:bat[:oid], X_64:bat[:oid]) :=
17         algebra.sort(X_17:bat[:str], false:bit, false:bit);
18     X_25:bat[:dbl] := sql.bind(X_4:int, "sys":str, "r":str, "a2":str, 0:int);
19     X_79:bat[:dbl] := algebra.projectionpath(X_63:bat[:oid],
20         C_5:bat[:oid], X_25:bat[:dbl]);
21     C_32:bat[:oid] := sql.tid(X_4:int, "sys":str, "s":str);
22     X_41:bat[:str] := sql.bind(X_4:int, "sys":str, "s":str, "v2":str, 0:int);
23     X_47:bat[:str] := algebra.projection(C_32:bat[:oid], X_41:bat[:str]);
24     X_34:bat[:str] := sql.bind(X_4:int, "sys":str, "s":str, "v1":str, 0:int);
25     X_40:bat[:str] := algebra.projection(C_32:bat[:oid], X_34:bat[:str]);
26     (X_67:bat[:str], X_68:bat[:oid], X_69:bat[:oid]) :=
27         algebra.sort(X_40:bat[:str], false:bit, false:bit);
28     (X_70:bat[:str], X_71:bat[:oid], X_72:bat[:oid]) :=
29         algebra.sort(X_47:bat[:str], X_68:bat[:oid],
30         X_69:bat[:oid], false:bit, false:bit);
31     X_55:bat[:dbl] := sql.bind(X_4:int, "sys":str, "s":str, "b2":str, 0:int);
```

```
32      X_81:bat[:dbl] := algebra.projectionpath(X_71:bat[:oid],
33          C_32:bat[:oid], X_55:bat[:dbl]);
34      X_83:bat[:dbl] := batcalc.+(X_79:bat[:dbl], X_81:bat[:dbl]);
35      X_18:bat[:dbl] := sql.bind(X_4:int, "sys":str, "r":str, "a1":str, 0:int);
36      X_78:bat[:dbl] := algebra.projectionpath(X_63:bat[:oid],
37          C_5:bat[:oid], X_18:bat[:dbl]);
38      X_48:bat[:dbl] := sql.bind(X_4:int, "sys":str, "s":str, "b1":str, 0:int);
39      X_80:bat[:dbl] := algebra.projectionpath(X_71:bat[:oid],
40          C_32:bat[:oid], X_48:bat[:dbl]);
41      X_82:bat[:dbl] := batcalc.+(X_78:bat[:dbl], X_80:bat[:dbl]);
42      X_76:bat[:str] := algebra.projection(X_71:bat[:oid], X_47:bat[:str]);
43      X_75:bat[:str] := algebra.projection(X_71:bat[:oid], X_40:bat[:str]);
44      X_73:bat[:str] := algebra.projection(X_63:bat[:oid], X_17:bat[:str]);
45      X_92:bat[:str] := bat.append(X_85:bat[:str], "sys.r":str);
46      X_94:bat[:str] := bat.append(X_87:bat[:str], "u1":str);
47      X_96:bat[:str] := bat.append(X_88:bat[:str], "clob":str);
48      X_98:bat[:int] := bat.append(X_89:bat[:int], 0:int);
49      X_100:bat[:int] := bat.append(X_91:bat[:int], 0:int);
50      X_101:bat[:str] := bat.append(X_92:bat[:str], "sys.s":str);
51      X_103:bat[:str] := bat.append(X_94:bat[:str], "v1":str);
52      X_105:bat[:str] := bat.append(X_96:bat[:str], "clob":str);
53      X_106:bat[:int] := bat.append(X_98:bat[:int], 0:int);
54      X_107:bat[:int] := bat.append(X_100:bat[:int], 0:int);
55      X_108:bat[:str] := bat.append(X_101:bat[:str], "sys.s":str);
56      X_109:bat[:str] := bat.append(X_103:bat[:str], "v2":str);
57      X_111:bat[:str] := bat.append(X_105:bat[:str], "clob":str);
58      X_112:bat[:int] := bat.append(X_106:bat[:int], 0:int);
59      X_113:bat[:int] := bat.append(X_107:bat[:int], 0:int);
60      X_114:bat[:str] := bat.append(X_108:bat[:str], "sys.L3":str);
61      X_116:bat[:str] := bat.append(X_109:bat[:str], "a1":str);
62      X_118:bat[:str] := bat.append(X_111:bat[:str], "double":str);
63      X_120:bat[:int] := bat.append(X_112:bat[:int], 53:int);
64      X_122:bat[:int] := bat.append(X_113:bat[:int], 0:int);
65      X_123:bat[:str] := bat.append(X_114:bat[:str], "sys.L3":str);
66      X_124:bat[:str] := bat.append(X_116:bat[:str], "a2":str);
67      X_126:bat[:str] := bat.append(X_118:bat[:str], "double":str);
68      X_127:bat[:int] := bat.append(X_120:bat[:int], 53:int);
69      X_128:bat[:int] := bat.append(X_122:bat[:int], 0:int);
70      language.pass(C_5:bat[:oid]);
71      language.pass(C_32:bat[:oid]);
72      language.pass(X_47:bat[:str]);
73      language.pass(X_71:bat[:oid]);
74      language.pass(X_40:bat[:str]);
75      language.pass(X_63:bat[:oid]);
76      language.pass(X_17:bat[:str]);
77      exit X_175:bit;
78      sql.resultSet(X_123:bat[:str], X_124:bat[:str], X_126:bat[:str],
79                    X_127:bat[:int], X_128:bat[:int], X_73:bat[:str],
80                    X_75:bat[:str], X_76:bat[:str], X_82:bat[:dbl],
81                    X_83:bat[:dbl]);
```

```
82  end user.s4_1;
```

Declaration, sorting, and addition of BATs are performed in lines 13-44. For example, lines 16-17, 26-27, and 28-30 illustrate sorting of $U1$, $V1$, and $V2$ attributes, respectively. Lines 34 and 41 show the addition between two pairs of BATs from the application parts.

3.8 Performance Evaluation

Setup All runtimes are averages over 3 runs on an Intel(R) Xeon(R) E5-2603 CPU (L1: 32+32K, L2:256K, L3:15360K), 1.7 GHz, 12 cores, no hyper-threading, 98 GB RAM, OS Debian Linux 4.9.189.

Competitors. We empirically compare the implementations of our relational matrix algebra (*RMA+*) with the statistical package *R*, the array database *SciDB*, and two state-of-the-art in-database solutions, *AIDA* [DDMK18] and *MADlib* [HRS+12]. (1) We implemented *RMA+* in MonetDB (v11.29.8) with two options for matrix operations: (a) BATs (RMA+BAT): No-copy implementation in the kernel of MonetDB; (b) MKL (RMA+MKL): Copy BATs to an MKL (v2019.5.281) [Int20] compatible format (contiguous array of doubles), then copy the result back to BATs. We execute linear operations (`add`, `sub`, `emu`) on BATs and use MKL for more complex operations. When the matrices do not fit into main memory we switch to BATs. Due to the full integration of RMA+, MonetDB takes care of core usage and work distribution, and all cores are used for relational and for matrix operations. (2) *SciDB* [SBPR11] uses an array data model, and queries are expressed in the high-level, declarative language AQL (Array Query Language) [Sci13]. SciDB uses all available cores. (3) *AIDA* is a state-of-the-art solution for the integration of matrix operations into a relational database and was shown to outperform other solutions like Spark or the pandas library for Python [DDMK18]. AIDA executes matrix operations in Python and offers a Python-like syntax for relational operations, which are then translated into SQL and executed in MonetDB (v11.29.3). AIDA uses all cores both in MonetDB and in Python. We also integrate our solution into MonetDB, which makes AIDA a particularly interesting competitor. (4) MADlib [HRS+12] (v1.10) provides a collection of UDFs for PostgreSQL (v9.6) for in-database matrix and statistical calculations. MADlib does not use multiple cores, which affects its overall performance. (5) The *R* package (v3.2.3) is highly tuned for matrix operations and is a representative of a non-database solution. R performs all relational operations

in data.tables structures and transforms the relevant columns to matrices to compute the matrix operations. An alternative approach to use character matrices for all operations is very inefficient (cf. Section 3.8.5). R uses all cores for matrix operations but runs relational operations on a single core.

Data. BIXI [Inc19] stores trips and stations of Montreal's public bicycle sharing system, years 2014-2017. DBLP [UoT19] stores authors with their publication counts per conference as well as conference rankings. The synthetic dataset used in the experiment to measure the effect of sparsity includes values between 0 and 5,000,000. All other synthetic datasets include real-valued numeric attributes with uniformly distributed values between 0 and 10,000.

3.8.1 Maintaining Contextual Information

A salient feature of our approach is that contextual information is maintained during matrix operations. We analyze the scalability of maintaining context and study an optimization that avoids sorting. To this end, we generate relations with a single application column and an increasing number of order columns. We compute `add` and `qqr` on these relations. Consider relations r and s with schemas $r(U1,...,U100,A1)$ and $s(V1,...,V100,B1)$, respectively. Attributes U_i and V_i form the order schemas of the first and second relation, respectively. The example RMA expressions for `add` and `qqr` are as follows:

```
1  SELECT * FROM ADD (r BY U1, ...., U100, s BY U1, ...., U100);
2
3  SELECT * FROM QQR (r BY U1, ...., U100);
```

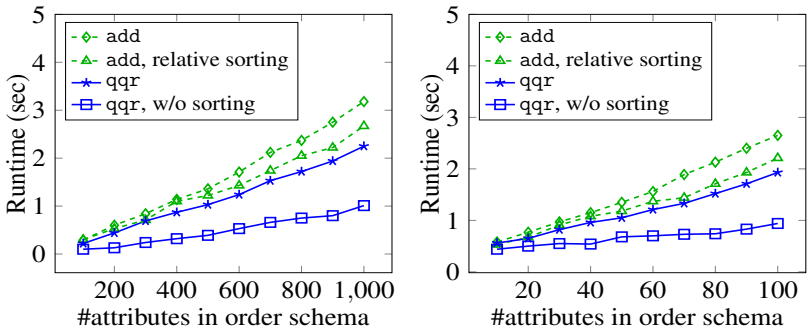
The first query adds the values of attributes $A1$ and $B1$ from, respectively, r and s ordered by the corresponding order schemas. The second query computes matrix Q from the QR decomposition applied to attribute $A1$ of relation r sorted by attributes $U1,...,U100$.

Since `add` and `qqr` are inexpensive for single column matrices, the main cost is the maintenance of the order part.

To handle contextual information we split, sort, morph, and merge lists of BATs (cf. Section 3.7.2). Sorting is the most expensive operation. Fortunately, sorting is not always necessary. For example, permuting the input rows for the `qqr` operation will affect the order of the result rows, but will not change their values. Therefore, sorting is not required. In element-wise opera-

tions like `add`, `emu`, or `sol`, only the relative order of the rows in the two input relations matters. Thus, only the order part of the second relation requires sorting (to get the same order).

Figure 3.15 shows the results. (1) Handling contextual information is efficient and scales to large numbers of attributes. (2) The optimized operators that (partially) avoid sorting clearly outperform their non-optimized counterparts.



(a) 100K tuples **(b)** 1M tuples

Figure 3.15: Handling contextual information

Note that a number of operations (`cpd`, `sol`, `rqr`, `dsv`, `tra`, `det`, `rnk`) do not preserve row context since the number of rows changes. Instead, a single column with predefined values (operation name or attribute names of the application schema) is created, which is negligible in the overall runtime.

3.8.2 Wide and Sparse Relations

Wide relations. Current databases scale better in the number of tuples than in the number of attributes. We test our RMA+ implementation in MonetDB on wide relations. We generate relations with 1000 tuples, one order attribute, and a varying number of application attributes. In Table 3.4, we increase the number of attributes from 1K to 10K and measure the runtime of the `add` operation. MonetDB can handle wide relations with several thousands of attributes, even though the runtime per column increases with the attribute number.

#attr	**1K**	**2K**	**3K**	**4K**	**5K**	**6K**	**7K**	**8K**	**9K**	**10K**
sec	0.6	2.2	4.8	8.8	13.4	20	27	36	47	62

Table 3.4: `add` over wide relations in RMA+

Sparse relations. We analyze the effect of MonetDB's built-in optimization on relations with many zeros. We add two relations (5M tuples, one order, 10 application attributes) with uniformly distributed non-zero values (range 1-5M). In Table 3.5 we increase the percentage of zero values (position of zeros is random) and measure the runtime: The `add` operation on sparse matrices is up to two times faster than the same operation on dense matrices. Thus, RMA+ leverages MonetDB's optimization features.

%	0	10	20	30	40	50	60	70	80	90	100
sec	1.68	1.60	1.49	1.41	1.33	1.25	1.16	0.99	0.94	0.89	0.76

Table 3.5: `add` over sparse relations in RMA+

3.8.3 RMA+ vs. Non-Database Approaches

We study the scalability of RMA+ to large relations and compare to *R* as a non-database solutions for matrix operations. In Table 3.6 we measure the runtime for `qqr` on tables with up to 100M tuples and 70 attributes in the application schema. For relations up to a size of 50Mx40, RMA+ delegates the matrix computation to MKL; the runtime includes copying the data. RMA+ is consistently faster than R since MKL can better leverage the hardware. R fails for sizes above 50Mx40 since it runs out of memory. In RMA+ we switch to the BAT implementation, which leverages the memory management of MonetDB. The Gram-Schmidt `qqr` baseline [Gan80] that we implemented over BATs is slower than the MKL algorithm (e.g., 834 vs. 61.4 sec for 50Mx40), which explains the increase in runtime. RMA+ scales to large relations that do not fit into memory (e.g., relation size 100Mx70 requires 56GB).

	10 attr		40 attr		70 attr	
System	**R**	**RMA+**	**R**	**RMA+**	**R**	**RMA+**
5M tup	3.5	2.1	20	6.6	47	11.6
50M tup	37	21.3	221	61.4	fail	2018
100M tup	74	40	fail	1690	fail	4064

Table 3.6: Runtimes of `qqr` in seconds in R and RMA+

3.8.4 RMA+ vs. Array Databases

We study the performance of RMA+ vs. SciDB [SBPR11] as a representative of array databases. We compute `add` on two matrices with 10 columns and a varying number of rows, followed by a selection.[4] The resulting runtimes are shown in Table 3.7. RMA+ outperforms SciDB by more than an order of magnitude. RMA+ performs addition directly over pairs of relations, while SciDB must compute a so-called array join [Sci13] over the input arrays in order to add their values.

#tuples	1M	5M	10M	15M
RMA+	4.6s	24.4s	1m18s	1m39s
SciDB	1m21s	7m6s	13m2s	18m23s

Table 3.7: `add` followed by a selection: RMA+ vs. SciDB

3.8.5 Overhead of Data Transformation

We investigate the overhead of data transformation for various matrix operations in a mixed relational/matrix scenario.

RMA+ is free to execute matrix operations directly on BATs or rearrange the numerical data in main memory and delegate the matrix operations to specialized packages like MKL [Int20]. R does not enjoy this flexibility: R uses the matrix data type for matrix operations and the data.tables storage structure for relational operations. While data.tables supports simple linear operations like linear model construction, the data must be transformed to the matrix type for more complex operations like `CPD`, `OPD`, or `MMU`. Matrices cannot store a mix of numerical and non-numerical values, which is required when working with tables; R offers character matrices, but they are very inefficient, e.g., joining trips and stations in the BIXI dataset takes 40 sec for the character matrix type and less than 2 sec for data.tables.

Figure 3.16 shows the percentage of time spent for data transformations on relations with 50 columns and a varying number of rows (100k to 500k). For R we measure the time of transforming the relation from data.table to matrix and back as a percentage of the overall query time, which includes the actual matrix operation. For RMA+ we measure the time share for copying the data from a list of BATs to a contiguous, one-dimensional array for MKL, and for copying

[4] We run this experiment on Ubuntu 14.04 since SciDB does not support Debian; Ubuntu runs on a server with 4 cores and 16GB of RAM.

the result back; the overall runtime in addition includes the matrix computation in MKL (but excludes the MonetDB query pipeline of query parsing, query tree creation, etc.).

#rows	(#columns = 50)					
500K	81	75	64	21	7	7
300K	79	77	63	21	7	7
100K	84	74	69	23	9	10
	ADD	EMU	MMU	QQR	DSV	VSV

(a) Data.table and matrix

#rows	(#columns = 50)					
500K	92	92	86	53	44	43
300K	91	91	86	55	45	40
100K	86	86	80	48	37	35
	ADD	EMU	MMU	QQR	DSV	VSV

(b) List of BATs and 1D array

Figure 3.16: Data transformation share: (a) R, (b) RMA+

Clearly, the overhead of transforming data matters for both R and RMA+. We draw the following conclusions: (a) Transforming data between data structures is costly.

(b) For simple operations like ADD and EMU, the transformation overhead dominates the overall runtime (up to 92%).

(c) For complex operations, the performance of the matrix operation dominates the overall runtime.

3.8.6 Efficiency for Mixed Workloads

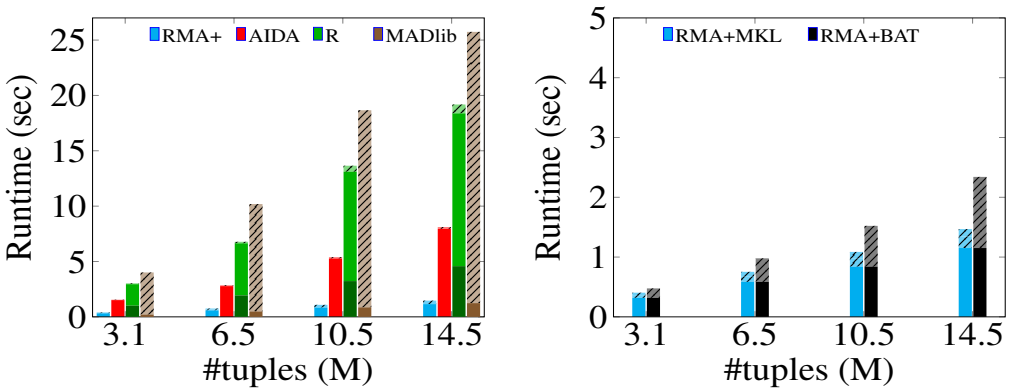
We analyze four workloads that require a mix of relational operations and matrix operations, and we compare our implementation of RMA (RMA+) to its competitors (R, AIDA, MADlib). The workloads stem from applications on our real-world datasets and differ in the complexity of relational vs. matrix part. On the BIXI dataset, we compute (1) the linear regression between distance and duration for individual trips, and (2) journeys connecting up to 5 trips; on DBLP we compute the (3) covariance between conferences based on the publication counts per conference and author; (4) on a synthetic dataset based on BIXI we count trips per rider.

(1) Trips – Ordinary Linear Regression Trips in BIXI include start date and start station, end date and end station, duration, and a membership flag for the rider; stations have a code, a name, and coordinates. At the level of relations, we need to perform the following data preparation steps: (a) Aggregate the trips and select those trips that were performed at least 50 times; (b) join trips and stations to retrieve the station coordinates and compute the distance. We use the OLS method [RRT95] to compute the linear regression between distance and duration. OLS uses cross product, matrix multiplication, and inversion: $\texttt{MMU}(\texttt{INV}(\texttt{CPD}(A,A)),\texttt{CPD}(A,V))$, where A is the matrix with the independent variables, and V is the vector with the dependent variable.

Figure 3.17a shows the runtime results for trips reported in the years 2014 (3.1M trips), 2014-2015 (6.1M trips), 2014-2016 (10.5M trips), and 2014-2017 (14.5M trips), respectively. The input data consists of numeric and non-numeric types such as date and time. We break the runtime down into data preparation (solid area of the bar) and matrix computation time (dashed light area) for RMA+, R, and AIDA; for R we also show the load time from a CSV file (dark area). RMA+ and AIDA outperform R and MADlib in all scenarios. R performs poorly on the relational operations of the data preparation step: The join implementation of R does not leverage multiple cores, and R lacks a query optimizer, which adversely affects the relational performance.

MADlib is outperformed by all other solutions due to the slow computation of the linear regression. RMA+ outperforms AIDA on all datasets. Although both RMA+ and AIDA compute the relational operations in MonetDB, RMA+ is up to 6.3 times faster: While AIDA passes pointers to access numerical Python data in MonetDB, this does not work for other data types (e.g., date, time, string) due to different storage formats [DDK19]. Therefore, expensive data transformations must be applied.

(2) Journeys – Multiple Linear Regression We compose trips that meet in a station into journeys. We start from 15M one-trip journeys of the form (start station, end station, duration); all attributes are numerical. During data preparation, we perform joins to create journeys of up to five trips, select those that appear at least 50 times, and join stations with their coordinates to compute the distances between subsequent stations in a journey. At the matrix level, we do a multiple linear regression analysis with the distances as independent variables and the overall duration as the dependent variable.

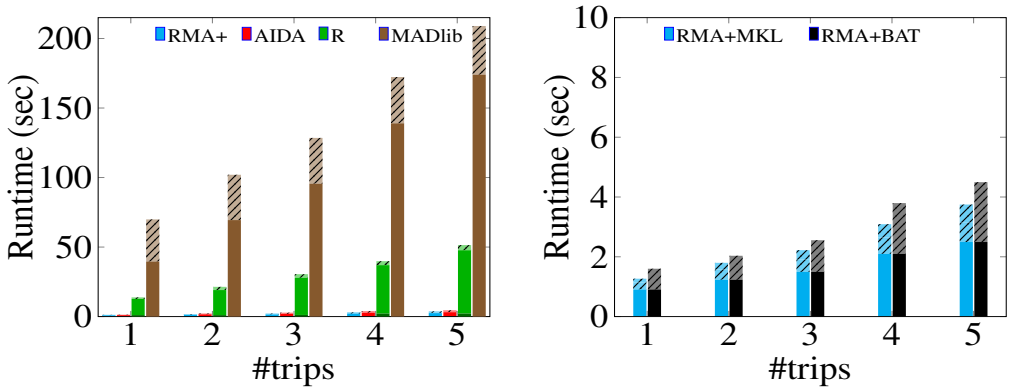


(a) System Comparison **(b)** RMA+BAT vs RMA+MKL

Figure 3.17: Trips (ordinary linear regression)

Figure 3.18a shows the runtime for journey lengths of 1 to 5 trips (i.e., 1 to 5 independent variables). The solid part is the time for data preparation (relational operations); the dashed light part is the time for multiple linear regression (matrix operations).

RMA+ and AIDA again outperform R on the relational part of the query. The relational part operates on purely numerical data and AIDA shows comparable join performance to RMA+. MADlib spends about two third of the relational runtime on distance computations and is therefore slower than its competitors also on the relational part.



(a) System Comparison **(b)** RMA+BAT vs RMA+MKL

Figure 3.18: Journeys (multiple linear regression)

(3) Conferences – Covariance Computation We compute the covariance between conferences with an A++ rating to lower rated conferences based on the number of publications per author and conference. The data includes two tables: *ranking* stores the rating (e.g., A++,

A+, B) for each conference whereas *publication* stores the number of publications per author and conference. Relation $publication(Author, AAAI, AABI, ...)$ is the result of SQL PIVOT over a count-aggregate by conference and author, where attribute *Author* represents authors names and the following attributes capture the number of publications in selected conferences. In relation $ranking(Conference, Rating)$ attribute $Conference$ represents conferences titles, and *Rating* captures their ratings.

Figure 3.19 illustrates the RMA expression that computes the covariance for conferences with an A++ rating to all conferences.

```
WITH t(Author, AAAI, AABI,  ...)  AS (
    SELECT *
    FROM SUB( publications BY Author,
              SELECT *
              FROM (SELECT Author FROM publications) AS a,
                   (SELECT AVG(Author) FROM publications) AS av
              BY Author)
    )
SELECT *
FROM CPD[C](t BY Author, t BY Author) AS tt
JOIN ranking ON tt.C = ranking.Conference
             AND ranking.Rating = 'A++' ;
```

Figure 3.19: RMA expression for covariance

Figure 3.20 illustrates the computation of the same task in R.

```
m1 = as.matrix(publication[1:ncol(publication)])
m2 = colMeans(m1);
m3 = m1 - m2;
m4 = crossprod(m3);
m5 = rownames(m4);
colnames(m5) = c('Conference')
m6 = cbind(m5, m4);
m7 = merge(m6, ranking, by.x = 'Conference', by.y = 'Conference')
m8 = subset(m7, Rating = 'A++')
```

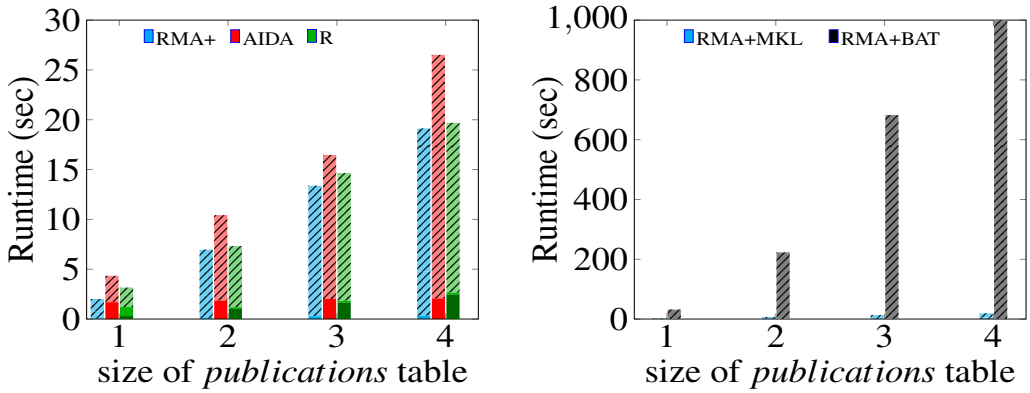
Figure 3.20: R expression for covariance

Note that the covariance computation in R (e.g., operation crossprod) does not return contextual information. In order to join the result with *ranking* and to select all A++ conferences, the conference names must be manually added as a new column.

We measure the runtime for *publication* tables of increasing sizes: (1) 337363x266 (i.e., 337363 authors and 266 conferences), (2) 550085x519, (3) 722891x744, and (4) 876559x882. The

ranking table stores 882 tuples. Note that the number of result rows of covariance is identical to the number of input columns, e.g., covariance of *publications* with 266 columns returns a relation (or matrix) of size 266x266.

Figure 3.21a shows the runtime results for RMA+, R, and AIDA. MADlib runs for 77, 429, 1086, resp. 1814 seconds on the different relation sizes and, thus, is omitted from the figure. In all systems, the covariance computation dominates the overall runtime with at least 90%. Since AIDA does not support covariance, we implement covariance via cross product [JW07] in all algorithms except MADlib, which has a `cov()` function but does not support cross product. For the cross product in RMA+ we use the routine `cblas_dgemm()`, in AIDA we use `a.t @ a`, in R we use `crossproduct`.

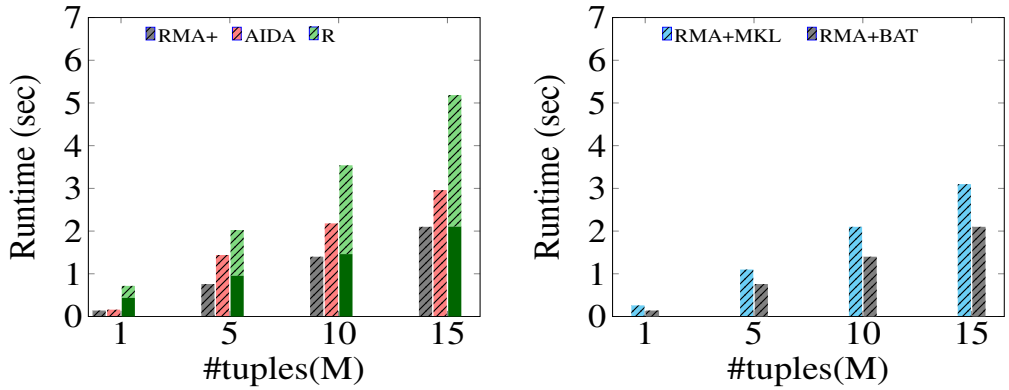


(a) Systems Comparison **(b)** RMA+BAT vs RMA+MKL

Figure 3.21: Conferences (covariance computation)

(4) Trip Count In Figure 3.22 we compute the number of trips per rider to 10 different destinations. Each tuple in the input relations stores a rider and the number of trips to each of the 10 locations for one year. We use `add` on the relations of two different years to get the trip count for a period of two years. We vary the number of riders from 1M to 15M and measure the runtime. Since `add` is a simple operation, RMA+ uses the no-copy implementation on BATs (RMA+BAT). RMA+ is faster than AIDA and R because it does not transfer data to Python (as AIDA) and does not translate data.tables to matrices (as R). MADlib takes 23, 119, 299, resp. 480 seconds for the different input sizes and, thus, is again omitted from the figure.

RMA+BAT vs. RMA+MKL Following our policy, RMA+ delegates matrix operations to MKL (RMA+MKL) in Figures 3.17a, 3.18a, and 3.21a (the operations are complex and we



(a) Systems Comparison **(b)** RMA+BAT vs RMA+MKL

Figure 3.22: Trip count (matrix addition)

do not run out of memory), and uses the no-copy implementation on BATs (RMA+BAT) in Figure 3.22a (`add` is a linear operation). We compare RMA+BAT to RMA+MKL in all scenarios. RMA+MKL outperforms RMA+BAT for the queries on trips (factor 1.8-3.8, cf. Figure 3.17b) and journeys (factor 1.4-1.9, cf. Figure 3.18b). For the conference query, RMA+MKL is 24 to 70 times faster since the cross product requires single element access and operates on relations with a large number of attributes. For the trip count, RMA+BAT outperforms RMA+MKL in all settings (cf. Figure 3.22b). Although elementwise addition is highly efficient in MKL, the transformation overhead cannot be amortized.

Syntactic query complexity We compare the query size (number of words) for the different systems. For ordinary linear regression, all systems require a similar number of words: RMA+: 114 words; R: 100 words; AIDA: 85 words. In case of multiple linear regression, the number of words for RMA+ is 154, for R is 148, and for AIDA is 96. RMA+ requires more words than AIDA due to its more flexible join syntax; R requires more words due to the renaming operations, which are required to avoid name collisions in the merge function. For the covariance computation between conferences, the RMA+ query uses 17 words, R 16 words, and MADlib 15 words. All three systems offer a dedicated covariance operation, and join and selection are performed on the covariance result.

3.8.7 Discussion

The key learnings from our empirical evaluation are the following: (1) RMA+ excels for mixed workloads that include both standard relational and matrix operations. (2) Only RMA+ can avoid data transformations in mixed workloads; data transformations may be costly and consume more than 90% of the overall runtime. (3) For complex matrix operations, however, transforming the data to a suitable format may pay off: In our approach, we are free to transform the data whenever beneficial. (4) In terms of scalability to large relations/matrices, our solution outperforms all competitors since it relies on the memory management of the database system for both the standard relational and the matrix operations. (5) Finally, the handling of contextual information, a feature of RMA, is efficient and can leverage optimizations that avoid expensive sortings.

CHAPTER 4

Building a System with Iterations

4.1 Introduction

In the past years the demand for analytical tasks performed over big data is increasing. More and more scientists state that there is a need for supporting complex linear computations inside database systems. Many important linear algebra operations are based on iterative computations [Var62]. Such linear operations perform iterations with a given exit condition over a matrix of a fixed size.

Currently there are two ways to perform iterations over relations: Recursive queries and UDFs. Recursive SQL queries use an on additive approach that adds new tuples in each step to the iterated relation (i.e., they use the UNION operation). This approach is far from the type of iterations required for iterative methods. UDFs can be leveraged for computing iterative methods inside a database system, but they are not integrated deeply into the optimization process of the DBMS. Unlike UDFs, we offer a solution that is integrated into the system and enables optimization for iterations.

Other state-of-the-art approaches introduce iterations on various levels. For example, new operations are introduced, where iterations are specific for the given operation, such as functions parametrized with lambda expressions [PTH+17], or iterations in CREATE TABLE statements [JLY+19]. These approaches address the lack of iterations for specific tasks, such as k-means and PageRank, without offering a principled solution. Our goal is to provide a solution that fits tasks that require iterations over a matrix of a fixed size. We want to integrate iterations into the relational model preserving its advantages, e.g., keeping available relational optimizations.

To integrate iterative methods into databases, we define and explore *shape preserving iterations* over relations. A shape preserving iteration is an in-place iteration. It works with an iterated relation, whose values are refined after each iteration. Shape preserving iterations have the following signature:

$$I_r(Q^r, E) = r'$$

They are based on *stable queries* (Q^r) and relational predicates E. Stable queries are the type of queries that can be used in shape preserving iterations. Predicates correspond to cost functions and quantify the quality of iterated relations. For example, the cost function in gradient descent measures how similar the estimation of the dependent variable values computed with the current iterated relation is to the given dependent variable values.

Shape preserving iterations require an input iterated relation to start with. Typically, values in such objects are created randomly and do not have a proper meaning until an iteration is over. We propose a solution to *randomly initialize* relations. This is a general approach that creates input relations with contextual information for shape preserving iterations. Shape preserving iterations extend the expressiveness of the relational matrix algebra [DAB20a] with a large set of analytical tasks based on iterative computations. We developed a simple syntax extension to integrate shape preserving iterations in SQL. Our extension is declarative and preserves existing optimization possibilities. We integrate and empirical evaluate shape preserving iterations into MonetDB to demonstrate the feasibility of our ideas.

Random initialization and shape preserving iterations are the key concepts that guarantee that *contextual information* in relations is preserved throughout the computation. Randomly initialized relations are created with contextual information, which is later preserved by shape preserving iterations. At the end of the computation, we get result relations that are interpretable and have *origins*.

We make the following technical contributions:

- We define *stable queries* and in-place *shape preserving iterations* over stable queries. Shape preserving iterations are an integral part of supporting iterative methods.
- We develop the concept of *randomly initialized relations* to add relations with random numeric values and meaningful contextual information to the relational model. Randomly initialized relations are input relations for shape preserving iterations. We prove that the set of relational matrix algebra operations is sufficient to create a randomly initialized relation with the proper contextual information.
- We prove that stable queries and shape preserving iterations over randomly initialized relations deliver result relations with sufficient contextual information, i.e., with origins.
- We integrate shape preserving iterations into the kernel of MonetDB. We explain the additions we made to the query processing pipeline to support shape preserving iterations. We compare our approach with a baseline implementation that flattens iterations.

The chapter is organized as follows. Section 4.2 includes the application scenario for our approach. In Section 4.3 we discuss related work. The terminology and existing approaches are introduced in Section 4.4. The concepts, definitions, and examples of stable queries, random initialization, and shape preserving iterations are given in Section 4.5, Section 4.6, and Section 4.7. The key properties of our approach are discussed in Section 4.8. Section 4.9 describes the implementation in MonetDB, and Section 4.11 reports the evaluation of our implementation.

4.2 Application Scenario

We focus on applications with iterations that require an iterated object with a stable shape. We consider logistic regression as an example of an iterative computation applied to relations to illustrate the type of tasks we target.

Consider relation p given in Figure 4.1 that includes information about patients. Tuple p_1 states that 39-year-old patient Clark smokes 0 cigarettes per day, has a total cholesterol level of 195 units, 26.97 body mass index, average glucose of 80 units, and his ten year risk of coronary heart disease (Y) is 0, i.e., it did not happen in the past ten years. Information about the risk of Y is binary and available for selected patients only.

p (patient)

	P	A	C	T	B	G	Y
p_1	Clark	39	0	195	26.97	80	0
p_2	Cox	46	0	250	28.75	95	0
p_3	Reed	48	20	245	25.37	75	NULL

Figure 4.1: Sample relation

The task is to estimate the risk of a coronary heart disease for patients for which the risk is unknown (tuples with NULL values in Y), i.e., we want to predict the probability of the disease (the prediction of Y must be between 0 and 1). The estimation is based on the information given in attributes A, C, T, B, and G of relation p. To solve the task, we use logistic regression, which is used to answer binary questions based on recorded observations. The prediction of Y is done in two steps: (a) the coefficients that quantify the impact of A, C, T, B, and G to Y are calculated with logistic regression, (b) the estimation of Y is calculated using the computed coefficients.

Logistic regression Consider matrices a and b, which include independent variables and an dependent variable, respectively. Logistic regression yields an approximate solution of x, which is the vector of coefficients (i.e., impact of the independent variables to the dependent variable) that satisfies the equation: $sigmoid(a * x) = b$, where $sigmoid(u)$ is the function $\frac{1}{1+e^{-u}}$. The sigmoid function maps the real values of u to the [0:1] interval.

In our case matrix a includes values of attributes A, C, T, B, and G, and vector b includes values of attribute Y. The logistic regression returns a relation with schema (V,Y), where attribute V contains for the names of the independent variables and attribute Y contains the respective coefficients.

Relation $v1$ in Figure 4.2a shows the computed logistic regression coefficients for the independent variables. For example, tuple $t1$ states that the impact coefficient of A on Y (i.e., of age on the coronary disease) is 0.34.

Risk estimation The estimation of Y is calculated using relational matrix multiplication [DAB20a]. The multiplication is performed between the independent variables and the logistic regression coefficients for people, whose risk is unknown, i.e., for tuples with a NULL value in Y attribute. After that, the estimation is mapped to the interval between 0 and 1 with the sigmoid function.

v1

	V	**Y**
t_1	A	0.34
t_2	B	-0.3
t_3	C	1.58
t_4	G	8.4
t_5	T	9.2

(a) Logistic regression coefficients

v4

	P	**Y**
z_1	Reed	0.8
...	...	...

(b) Estimated risk

Figure 4.2: Result relations

The relational algebra expression in Figure 4.3 computes the estimation of attribute Y.

$$v2 = \pi_{P,A,B,C,G,T}(\sigma_{Y\ IS\ NULL}(p))$$
$$v3 = \mathtt{mmu}_{P;V}(v2, v1)$$
$$v4 = \pi_{P,(SIGMOID(Y))}(v3)$$

Figure 4.3: Computing prediction of Y

Result relation $v4$ shown in Figure 4.2b includes the estimation of risk of the coronary disease. For example, tuple z_1 states that Reed has an 80% chance of having the coronary heart disease in the next ten years.

4.3 Related Work

Passing et al. [PTH+17] offer iterations over relations in HyPer. The approach introduces iterations on two levels: Iterations that are hidden in new operations, such as k-means or PageRank, and SQL level iterations that are available for a user to program. The first type of iterations is specific for each operation algorithm and is highly adjusted to HyPer. The second type of iterations is a more general approach and is conceptually similar to our idea. However, we precisely define, study properties, and explain the integration in a column-store of shape preserving iterations, while the details of the implementation and properties of the introduced iterations in [PTH+17] are not described.

Jankov et al. [JLY+19] extend SimSQL with arrays, whose elements are relations, in order to incorporate neural networks into the relational model. Each table is defined through a query over

previously defined tables, enabling recursive table definitions. Then, each step of the iteration creates a new table, rather than updating the existing table as in our approach. As a result the optimizer must handle enormously large query plans. The approach focuses on how to cut plans into pieces, which are optimized and executed independently. This is an NP-hard task, which leads to greedy heuristics to find an approximate solution.

Binnig et al. [BRFR12] offer an SQL extension with functions that support recursive iterations and table assignment operations to bring interactivity and procedural flavour to SQL. New features are implemented with the help of graphs with cycles. Since iterations are available only in functions, this approach does not support a full integration into the SELECT statement. Binnig offers simple optimization such as push downs of selections and projections in the new query graphs.More advanced optimization techniques, e.g., reordering joins, have not been developed. The optimization of cycles in query graphs is not discussed.

Standard recursive queries in SQL are based on iterations. After each iteration step, the resulting tuples are added to the iterated relation until the iteration step returns an empty set of tuples. This approach can be used for computation of iterative methods over relations. However, the recursive SQL solution stores all intermediate results in the iterated relation. This leads to a poor time and space efficiency due to the preservation of a large amount of tuples that are not needed. Recursive queries do not preserve the shape of an iterated relation, and they are not suitable for shape preserving iterations we want to integrate into a database.

4.4 Background

Relation A relation r is a set of tuples with schema $sch(r)$. A schema, $sch(r) = (A, B, \ldots)$, is a finite, ordered set of attribute names. The underlined attributes (e.g., $\underline{C}, \underline{D}$) form a key. $|r|$ is the number of tuples in relation r, and $\#r$ is the number of attributes in relation r. A tuple of r has a value from the appropriate domain for each attribute in the schema.

Iteration An iteration over relations takes relations as input and modifies one of the input relations in each step. The modified relation is called an iterated relation, and is returned as the result when the iteration has finished. An iteration has an iteration body and an exit condition. The iteration body is repeated until the exit condition evaluates to true.

Relational Matrix Algebra We leverage the extension of the relational algebra with relational matrix operations (RMA), which supports basic matrix operations, such as multiplication and inversion, over relations [DAB20b, DAB20a].

For all relational matrix operations input and result relations are composed of two parts: Contextual information and an application part. The contextual information identifies and describes each cell in the application part. Input contextual information includes an ordering part (i.e., values of attributes responsible for determining the order of tuples) and ordering and application schemas. Input application part is used in the corresponding matrix operation. Result contextual information consists of row and column origin defined in [DAB20b]. Origins are inherited from input contextual information and, thus, connect input and result relations. The inheritance is based on the shape of a result relation. Consider relation r with origins, then the row origin of r is denoted as $ro(r)$ and the column origin of r is denoted as $co(r)$. Figure 4.4 illustrates the structure of input and result relations: White cells are the contextual information, gray cells are the application part.

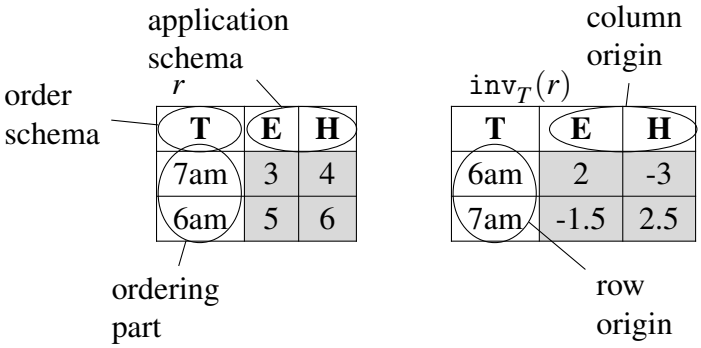


Figure 4.4: Structure of input and result relations in RMA

In relational matrix operations, each input relation r is accompanied by a list of attributes **U** called an order schema. The order schema is part of the contextual information and determines the order of tuples for the matrix operation. The rest of the attributes in r, $sch(r) \setminus \mathbf{U}$, is called an application schema. For example, the relational matrix inversion over relation r from Figure 4.4 is expressed as follows:

```
1  SELECT * FROM INV(r BY T);
```

Here, T is the order schema that determines the order of tuples for the inversion.

Logistic regression The compuatation of logistic regression is based on iterative gradient decent: In each step the iteration refines the coefficients to minimize the cost function [Rud16]. Typically, the gradient descent computation is parametrized with a threshold and stepsize. The stepsize defines the size of changes for coefficients in each iteration. The iteration stops when the error is smaller than the given threshold.

Given the logistic regression formula $sigmoid(a*x) = b$, threshold ε, and stepsize α, Algorithms 5 and 6 illustrate the gradient descent iteration. Lines 4-6 in Algorithm 5 refine the coefficients as long as the cost function computed in Algorithm 6 is bigger than the threshold (line 7 in Algorithm 5).

Algorithm 5: LogRegIteration(a, b, x, ε, α)

```
1 initialize(x);
2 m = b.Length();
3 repeat
4 |   k = a*x;
5 |   h = SIGMOID(k);
6 |   x = x − α ∗ a^τ ∗ (h − b)/m ;
7 until cost(x,a,b) < ε ;
8 return x ;
```

Algorithm 6: cost(x, a, b)

```
1 m = b.Length();
2 k = a*x;
3 h = SIGMOID(k);
4 c = −b ∗ log(h) − (1 − b) ∗ log(1 − h) ;
5 c = c.Sum()/m ;
6 return c ;
```

Example 26. Consider the computation of gradient descent with Algorithms 5 and 6 applied to the matrices corresponding to the relation from our application scenario. Figure 4.5 illustrates the first two steps of the iteration, i.e., it illustrates the input matrices and the intermediate results of Algorithm 5. We assume that α is 0.001 and ε is 0.1. Matrix a includes the independent variables, i.e., values of attributes A, B, C, G, and T from relation p (Figure 4.1); matrix b includes the dependent variable, i.e., values of attribute Y, and the initially guessed coefficients in matrix x are generated randomly.

First, the computation of the iteration body, i.e., lines 4-6 from Algorithms 5, is shown. We compute matrices k and h and refine matrix x by subtracting the gradient multiplied by the stepsize

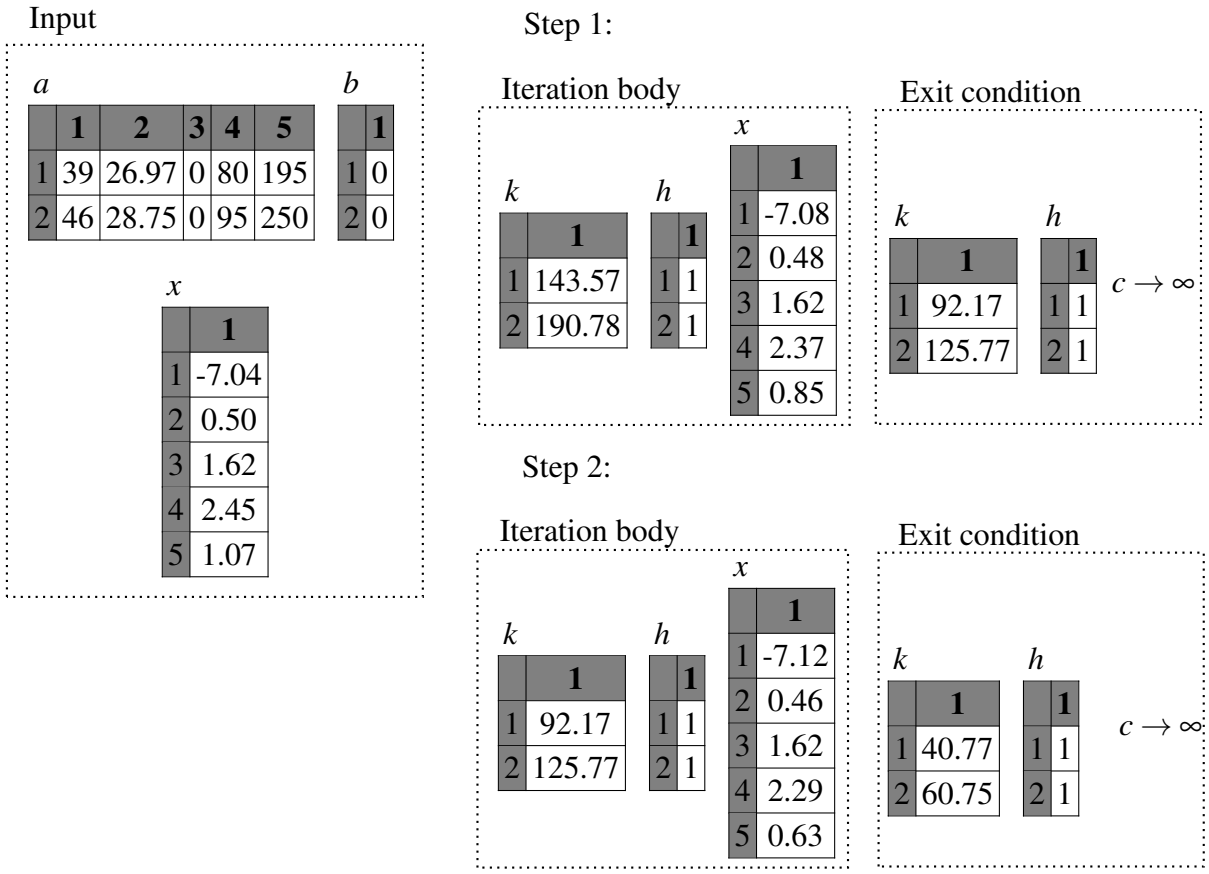


Figure 4.5: Gradient descent computation steps

$(\alpha * a^{\tau} * (h-b)/m)$ from x. Matrix k and matrix h contain the non-normalized, i.e., before applying the sigmoid function, and normalized estimations of the dependent variable, respectively. Thus, the values in matrix h are the risk estimation with the current coefficients in matrix x. Then the exit condition for the current x is evaluated. The cost function computes how close the estimated risk values in h to the real dependent variable values given in b. The cost on the fist step is close to infinity ($c = -log(1-h) = -log(0) \to \infty$) because both values in h are predicted wrongly: They are ones, and the real risk values in b are zeros. As the cost is bigger than threshold 0.1, the iteration repeats the iteration body and evaluation of the exit condition in the second step. The values in x are refined and the cost is checked again. The updated estimation is still incorrect for both patients (i.e., both values in h are ones) and the cost function is again close to infinity. However, the values in matrix k are smaller than in the previous step. Therefore, values in x are closer to the correct result, because the smaller values in k, the closer the values in h are to zero.

After that the iteration body is repeated again, until the cost is smaller than threshold 0.1. Throughout the iteration, coefficients, i.e., the values in x, are replaced with the result values returned by the iteration body. □

4.5 Stable Queries

As a first step towards bringing iterative methods to the relational model, we introduce *stable queries*. Iterative methods keep the size of an input matrix unchanged throughout the computation. Stable queries are relational algebra expressions with input and result relations of the same size.

Definition 7. *(Stable Query)* Consider relational algebra expression $Q(r, r1, r2, ...)$ that returns result relation r'. Q is a stable query for r if r and r' have the same number of tuples and the same number of attributes, r and r' have a set of common attributes with the same values, and the composition of these attributes is a superkey:

$$|r| = |r'| \wedge \#r = \#r' \wedge \exists A \subset sch(r), sch(r') \\ (r.A = r'.A \wedge A \textit{ is a superkey})$$

A stable query Q for r is denoted by Q^r.

Example 27. (Stable Queries) Given relation r with schema $(\underline{A}, B, C)$ and numeric attributes B and C. Consider the following relational matrix algebra expressions: $\pi_{A,B,C+10}(r) = v1$ and $\mathtt{qqr}_A(r) = v2$. Both expressions are stable queries for r. The first expression returns result relation $v1$ with the same number of tuples as in relation r. The result and input relations have common attributes A, B with the same values. Since A is a primary key, (A, B) is a superkey. The second expression is stable because $v2$ has the same shape as r, and $v2$ and r have common key attribute A. □

Example 28. (Not Stable Queries) Given non-empty relations r and s with schemas $(\underline{A}, B, C)$ and $(\underline{A}, D)$, respectively. Attribute C is numeric. Consider the following relational algebra expressions: $\pi_{C*2,C+10}(r) = v3$ and $r \bowtie s = v4$. The first expression returns result relation $v3$ with the same number of tuples as in relation r. However, r and $v3$ do not have any common attributes with the same values. Hence, this is not a stable query. The second query is not stable for r or s because the number of attributes in result relation $v4$ is equal to four, and, thus, $\#r \neq \#v4$ and $\#s \neq \#v4$. □

4.6 Random Initialization

Typically, iterative methods iterate over input matrices that are initialized with random values. Matrix x in Figure 4.5 is a typical example of an iterated matrix with randomly generated values. Currently SQL does not provide functionality to create relations with random values in the application part and the correct contextual information. We introduce randomly initialized relations: The relations, that inherit contextual information from existing relations.

Definition 8. *(Randomly Initialized Relation)* Consider a set of relations $S = (r_1, ..., r_n)$. r is a *randomly initialized relation* if (1) its contextual information consists of origins that are inherited from relations in S and an order schema:

$$\exists r_i, r_j \in S(ro(r) \in r_i \wedge ro(r) \in r_j)$$

and (2) values in the application part are randomly generated.

Each randomly initialized relation consists of an application part and contextual information. Thus, these relations are suitable as input relations for RMA expressions. The contextual information of a randomly initialized relation is the combination of application schemas and order parts of relations in S.

A randomly initialized relation has the structure shown in Figure 4.6. It includes the row origin, the column origin, and the application part. Contextual information values that belong to the row or column origins are inherited from existing relations. Throughout an iteration the values in the contextual information remain the same and the values in the application part are refined.

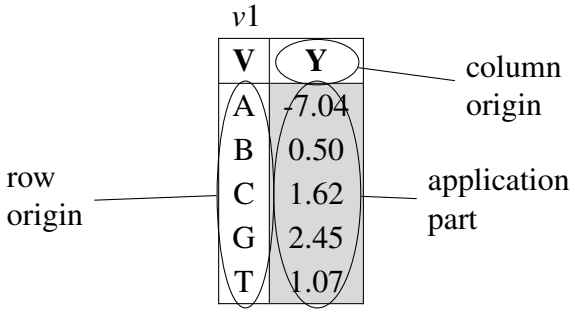


Figure 4.6: Structure of a randomly initialized relation

The values in the application part of a randomly generated relation are an initial guess of the result values. The contextual information of a randomly initialized relation describes the final values rather than the initial ones.

Example 29. Consider relation $v1$ from Figure 4.6. This is a randomly generated input relation for logistic regression, whose contextual information is inherited from relation p shown in Figure 4.1. The contextual information does not truthfully describe the random values in the application part. For example, values 'C' and Y describe that the cell with the value 1.62 contains the impact of one cigarette per day to the yearly risk of the coronary disease. The value 1.62 is far from the real impact and is refined on each step of logistic regression. Figure 4.2a shows the result relation $v1$ with the estimated final values in the application part. The contextual information stays the same throughout the computation and describes the result application part values. □

4.7 Shape Preserving Iterations

In this section we define iterations used in iterative methods over relations. We call such iterations *shape preserving iterations*, since the number of attributes and tuples of an iterated relation remains stable and only its values are refined.

An iteration body and an exit condition of a shape preserving iteration are relational expressions. The iteration body is a stable query for the iterated relation. A shape preserving iteration executes its iteration body and replaces the values in the iterated relation with the values that are returned by the iteration body. Then the exit condition is executed, and in case the condition evaluates to false, the iteration body is repeated again.

A shape preserving iteration stops when the exit condition evaluates to true. The exit condition is a relational predicate, which corresponds to a typical exit condition of an iterative method, i.e., when a cost function reaches a threshold. Thus, the iteration stops when the values in the iterated relation are close enough to the optimal values. There are two ways to determine the proximity to the optimal values: (1) calculate the cost function based on the iterated relation and the input values and compare it with the threshold, (2) calculate the distance between two consecutive states of the iterated relation and compare it with the threshold. For example, the first option is used in regression computations, where the cost function is based on the estimated and real dependent variable values, and is compared with the threshold. The second option is used, for example, in k-means clustering algorithm, where the distance between two consecutive

positions of cluster centers is calculated. In this case the exit condition of a shape preserving iteration includes the computation of the next state of the iterated relation.

Definition 9. *(Shape Preserving Iteration)* Consider iteration I with iterated relation r, iteration body Q, and exit condition E. I is a shape preserving iteration if Q is a stable query for r and E is a relational predicate:

$$I_r(Q^r, E) = r'$$

Shape preserving iterations are in-place iterations, meaning that after each execution of the iteration body, values in r are replaced with the refined values, returned by the iteration body. Result relation r' is equal to relation r with refined values in the application part.

By definition of a stable query, r and r' have the same size and a set of common attributes. Thus, an iterated relation includes contextual information, i.e., its schema and a set of attributes with common values, which remain the same throughout the computation; and an application part, whose values are refined by a shape preserving iteration. The preservation of contextual information in iterated relations guarantees the presence of contextual information in result relations, and, thus, their interpretability. Intuitively, the application part of an iterated relation corresponds to an iterated matrix in iterative methods. As the size of the application part and the meaning of its values do not change through the computation, the contextual information stays valid in the result relation. Shape preserving iterations are fixed point iterations, where the number of performed steps depends on the values in iterated relations.

Note that SQL recursive queries are not shape preserving iterations and cannot be expressed in the given notation. The iteration body in an SQL recursion has the following structure:

```
1 Q1 UNION ALL Q2.
```

This query is not stable for the relation that is recursively computed. An exit condition in SQL recursive queries is not given explicitly while in our approach Q^r and E are independent of each other.

Example 30. (Shape Preserving Iteration) Given relation s with schema $(\underline{A}, B, C)$ and numeric attributes B and C. Consider iteration $I_s(\pi_{A,B,C/2}(s), \vartheta_{SUM(C)}(s) < 1)$. This is a shape preserving iteration: The iteration body Q is a stable query, because the values in attributes A and B as well

as the number of tuples are preserved throughout the computation. The values in relation s are updated until the sum of values in attribute C is smaller than 1. □

4.7.1 Applications

Shape preserving iterations are used in iterative methods, that include variations of gradient descents, approximations of linear systems, and data clustering algorithms. Another example of an iterative method is successive overrelaxation used in support vector machine algorithms to perform multiclass classifications [MM99]. Many real-world applications for iterative methods, such as sonar, radar, and astronomy array processing or medical tomography, are described in [Byr08]. We consider two representative use-cases in detail.

Linear Approximation Linear approximation is used to solve linear matrix equations in the form of $m * x = n$, where m and n are given matrices, and x is a matrix that must be found. Although there exists a closed form solution, e.g., $x = m^{-1} * n$, the computation of the inverse matrix might be too expensive of m is big. Thus, when the exact solution is not a necessity and can be approximated, iterative methods are preferred because they are computationally less expensive. The Gauss-Seidel method [Bla06] is one of the linear approximation techniques. It works with $m = m_1 + m_2 + m_3$, where m_1 is the strictly lower triangular part of m, m_3 is the strictly upper triangular part of m, and m_2 is the diagonal part of m. In the beginning it initializes matrix x with random values, which are usually close to zero. The iteration body of the iteration in Gauss-Seidel approach is $x_{i+1} = m_2^{-1} * (n - (m_1 + m_3) * x_i)$, and the exit condition is $||m * x_i - n|| < \tau$. As only the diagonal values in m_2 are non-zero values, m_2^{-1} is a cheap operation, compared to computing m^{-1}.

Example 31. Consider matrix equation $m * x = n$. Figure 4.7 illustrates the input matrices and the two first steps of the linear approximation applied to matrices m, n, and iterated matrix x, which is populated with small random values. The right part of Figure 4.7 contains the intermediate results of the iteration body and the computation of the exit condition. Matrix k corresponds to the result of $m_2^{-1} * n$, matrix h is $m_2^{-1} * (m_1 + m_3) * x$, and matrix p is the prediction of n with the current values of x, i.e., $m * x$. The size of x remains the same throughout the iteration, and its values are replaced with refined values in each step. □

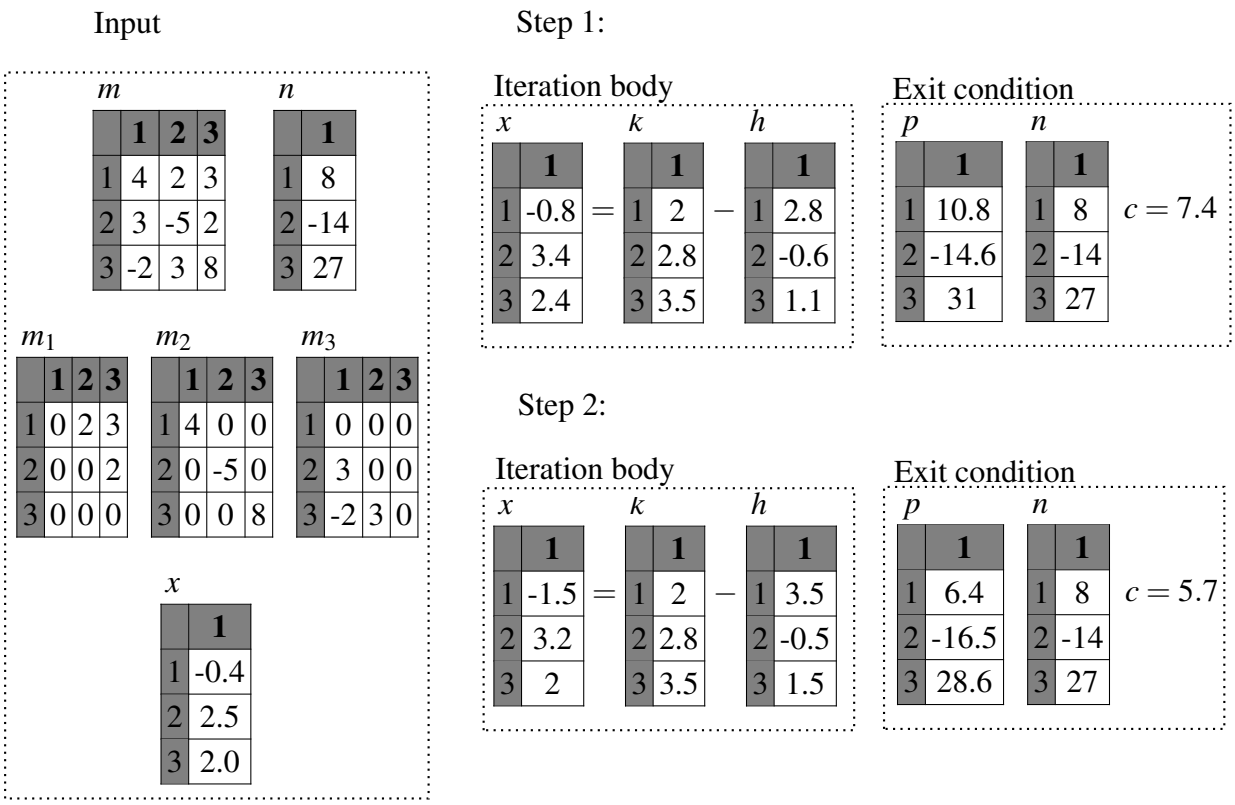


Figure 4.7: Linear approximation computation steps

K-means The k-means algorithm partitions observations into clusters. The algorithm takes matrix m with observations and iterates over matrix x that contains the centers of the clusters. The iteration over x is performed until the cluster centers remain unchanged.

The iteration body is $x_{i+1} = center(min(m, x_i))$, meaning that the new centers of the clusters are computed based on the partition found on the previous step. The exit condition is $||x_{i+1} - x_i|| < \varepsilon$.

Example 32. Consider matrix m and matrix x shown in the input box in Figure 4.8. Matrix m stores the coordinates of five points in 2D space that should be grouped in two clusters, and matrix x stores the initially guessed coordinates of cluster centers. Figure 4.8 illustrates the steps of the k-means algorithm applied to m, x, and threshold 0.1. The iteration body assigns the points in m to a cluster with the nearest center. The intermediate result of the iteration body computation, matrix n, stores points coordinates along with the coordinates of the nearest center. Thus, on the first step points 1, 2, and 4 are assigned to the first cluster, and points 3 and 5 to the second cluster. The iteration body returns the new coordinates for each cluster based on the coordinates of its points. The exit condition compares the two consecutive states of the clusters centers. To do that, the new values in x must be computed (i.e., the exit condition includes the

iteration body logic). On the first step the distance between the new and old centers is 2.9, which is bigger than threshold 0.1, and the iteration body is repeated again. In the second step the new clusters centers have the same coordinates as old centers, and the iteration stops.

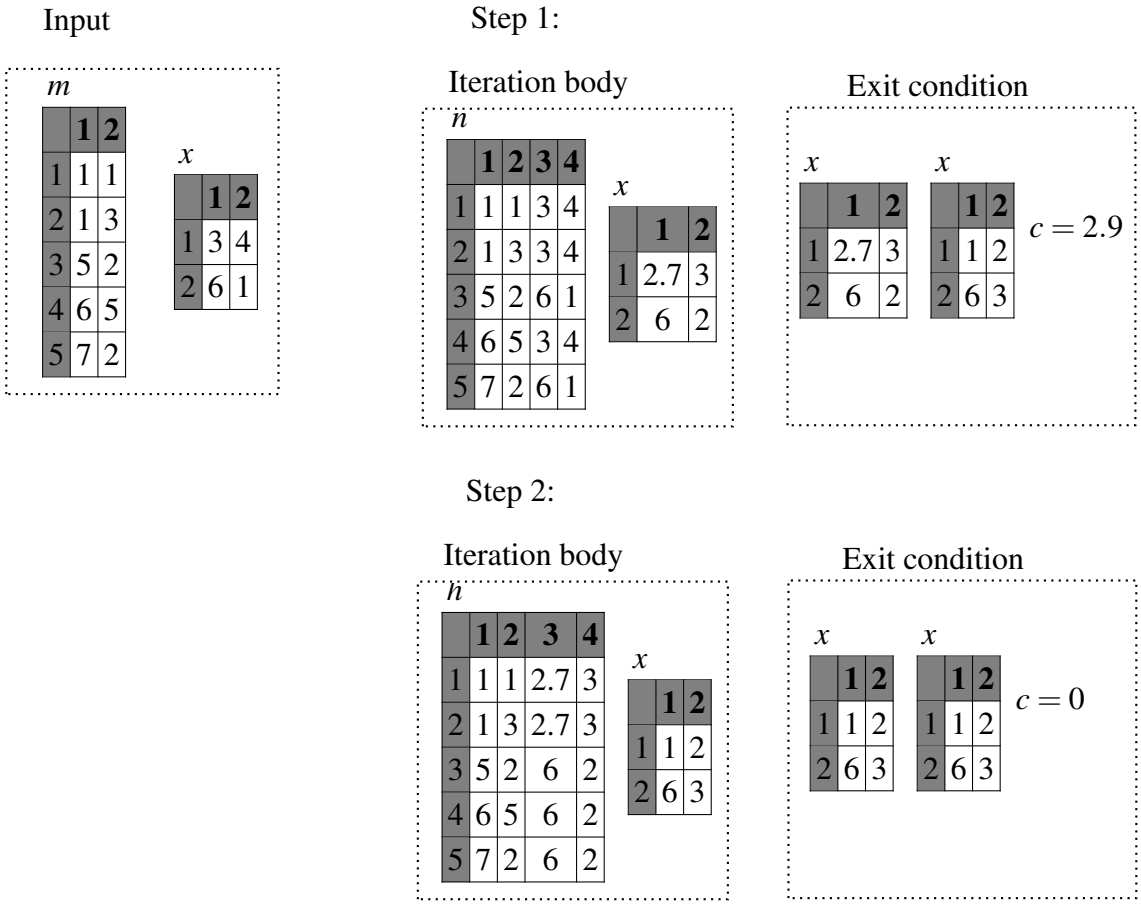


Figure 4.8: K-means computation steps

□

The described algorithms use iterations with an iteration body and an exit condition. In general, the iteration body computes a certain function taking the iterated matrix as an input. For example, it can compute a subtraction, multiplication, or gradient. The exit condition computes a cost function based either on the current iterated matrix or on two consecutive states of the iterated matrix. The main assumptions of iterative methods are that the iterated matrix has a fixed size, the iteration converges, and the optimal solution (i.e., the matrix with the optimal values) exists. These assumptions also hold true for shape preserving iterations.

There are tasks that require iterations and cannot be solved with shape preserving iterations. For example, a query to find all ancestors of a given person in a hierarchical data set. Such queries operate with sets rather than with numbers, and the shape of the iterated object grows as the iteration progresses.

4.7.2 Logistic Regression with Shape Preserving Iterations

In this section we explain our solution for the first step of the application example, i.e., the computation of the coefficients of logistic regression. We develop a shape preserving iteration that performs gradient descent to find the coefficients.

Logistic regression is computed over attributes A, B, C, G, T, and Y from relation p. Attributes A, B, C, G, and T are the independent variables that correspond to matrix a. Attribute Y is the dependent variable that corresponds to vector b. The iterated relation $v1$ is randomly initialized. $v1$ has two attributes: V with the names of the independent variables and Y with the respective coefficients of the logistic regression.

The shape preserving iteration for the gradient descent has the following form: $I_{v1}(Q^{v1}, E)$, where Q^{v1} is the iteration body corresponding to Algorithm 5, and E is the exit condition corresponding to the comparison of the cost function from Algorithm 6 with a threshold. Figure 4.9 illustrates the relational matrix expressions [DAB20a] for the iteration body and the exit condition, i.e., Q^{v1} and E, respectively.

Input

$v1$
$p1 \leftarrow \pi_{P,A,B,C,G,T}(p)$
$p2 \leftarrow \pi_{P,Y}(p)$
$t \leftarrow \vartheta_{COUNT(P)\rightarrow N}(p2)$

Iteration body

$e \leftarrow \pi_{P,SIGMOID(Y)\rightarrow YE}(\mathtt{mmu}_{P;V}(p1, v1))$
$t1 \leftarrow \pi_{P,(Y-YE)\rightarrow Y}(p2 \bowtie e)$
$t2 \leftarrow \pi_{P,Y/N\rightarrow YG}(t1 \times t)$
$g \leftarrow \mathtt{cpd}^{V}_{P;P}(p1, t2)$
$Q^{v1} : \pi_{V,(Y-\alpha*YG)\rightarrow Y}(v1 \bowtie g)$

Exit condition

$E : \pi_{S/N}((\vartheta_{SUM(Y*log(YE)-(1-Y)*(log(1-YE)))\rightarrow S}(e \bowtie p2)) \times t) < \varepsilon$

Figure 4.9: The shape preserving iteration for logistic regression

The input box includes the randomly initialized relation $v1$ and relations $p1$, $p2$, and t, which illustrated in Figure 4.10. The auxiliary relations $p1$, $p2$, and t simplify the iteration body and the exit condition. Relations $p1$ and $p2$ are the results of projections of p on the independent and dependent variables. Relation t stores the number of tuples in relation $p2$ and is used for normalizing the gradient and the cost function.

Input:

$p1$

P	A	C	T	B	G
Clark	39	0	195	26.97	80
Cox	46	0	250	28.75	95

$p2$

P	Y
Clark	0
Cox	0

t

N
2

Iterated:

$v1$

V	Y
A	-7.04
B	0.50
C	1.62
G	2.45
T	1.07

Figure 4.10: Input relations for logistic regression in application scenario

The iteration body includes the steps of query Q^{v1} computation with the intermediate result relations e, $t1$, $t2$, and g. Relation e stores the estimation of the risk based on the current coefficients. Relation $t1$ stores the difference between the estimated and the real dependent variable values. Relation $t2$ stores the normalized difference. Relation g stores the gradient values. After each computation of the iteration body, the values of $v1$ are updated. Exit condition E computes the cost function and compares it with threshold ε.

The iteration body and updates of $v1$ are repeated until the cost function is smaller than ε. The shape preserving iteration yields result relation $v1$ shown in Figure 4.2a.

Example 33. Figure 4.11 illustrates the computation of the iteration body in the first step of the shape preserving iteration from Figure 4.9. It includes input relation $v1_{old}$, gradient relation g, and refined relation $v1_{new}$. Once $v1_{new}$ is computed, the predicate in the exit condition is evaluated. If it evaluates to false, the iteration body is repeated again for relation $v1_{new}$ with the refined coefficients.

□

Many iterative methods have variations, such as using different cost functions or metrics. For example, a regularization technique [Ng04] might be required in the gradient descent computation to avoid overfitting. Our approach allows to adjust the iteration body and the exit condition according to the requirements of the corresponding iterative method.

$v1_{old}$

V	Y
A	-7.04
B	0.50
C	1.62
G	2.45
T	1.07

g

V	Y
A	42.5
B	27.86
C	0.00
G	87.5
T	222.5

$v1_{new}$

V	Y
A	-7.08
B	0.48
C	1.62
G	2.37
T	0.85

Figure 4.11: One step of the iteration with $\alpha = 0.001$ in the application scenario

4.7.3 Syntax Extension

In order to support shape preserving iterations we extend the `WITH` clause. Figure 4.12 illustrates the syntactic construction for iterations.

```
1 WITH
2   ITERATED r(A1, A2, ...)
3     INITIAL (R)
4     AS (Q)
5   UNTIL P
6 SELECT * FROM r;
```

Figure 4.12: Iterations in SQL

Relation r is the iterated relation, which is initialized with the result of query `R`. Query `Q` corresponds to the iteration body and computes the new values, and `P` is the predicate that specifies the exit condition. The values in relation r are updated after each execution of query `Q`. Thus, `Q` is repeated and relation r is updated until `P` evaluates to true. Since we target shape preserving iterations, query `Q` must be a stable query for relation r.

Example 34. Figure 4.13 shows the SQL expression that corresponds to shape preserving iteration $I_s(\pi_{A,B,C/2}(s), \vartheta_{SUM(C)}(s) < 1)$ given in Example 30. The iterated relation s is initialized with relation r given in Figure 4.4.

The query after INITIAL initializes the iterated relation s. The part of the SQL query between AS and UNTIL corresponds to iteration body $Q^s = \pi_{A,B,C/2}(s)$. The predicate after UNTIL is exit condition $E = \vartheta_{SUM(C)}(s) < 1$. □

```
1  WITH
2    ITERATED s(A, B, C)
3      INITIAL (SELECT * FROM r)
4      AS (
5        SELECT A, B, C/2
6        FROM s
7        )
8    UNTIL
9      SELECT SUM(C)
10     FROM s
11     <
12     1
13 SELECT * FROM s;
```

Figure 4.13: Example of a query with shape preserving iteration

4.8 Properties of Random Initialization and Shape Preserving Iterations

In this section we discuss the properties of shape preserving iterations that iterate over randomly initialized relations. We show that randomly initialized relations with proper contextual information can be created with relational matrix expressions. We prove that shape preserving iterations yield result relation with sufficient contextual information.

4.8.1 Random Initialization

Lemma 1. *Consider a set of relations S, and relation r that inherits origins from relations in S. There exists a relational matrix algebra expression that returns r', such that r and r' differ only in the application part values.*

Proof. Relation r inherits origins from relations in S. Thus, the row origin of r is an order part or an application schema inherited from a relation in S, and the column origin of r is an order part or an application schema inherited from a possibly different relation in S. We show that a relational matrix algebra expression exists for each possible combination of inheritance of r's origins.

We use the shape type notation of matrix operations introduced in Dolmatova et al. [DAB20a]. The shape type defines how the result cardinalities are inherited from the input cardinalities. For example, matrix multiplication (MMU) has shape type (r_1,c_2). This means that the number of result rows (i.e., r_1) is inherited from the number of rows in the first input matrix, and the number of result columns (i.e., c_2) is inherited from the number of columns in the second input matrix. We

use the same notation for the corresponding relational matrix operations, e.g., mmu has the same shape type as MMU.

Let us consider two cases: (1) origins are inherited from one relation s_1; (2) origins are inherited from two distinct relations s_1 and s_2.

(1) Inheritance from one relation. There are four combinations of origins inheritance: (r_1,r_1), (r_1,c_1), (c_1,c_1), and (c_1,r_1). Therefore, the result relation can be created with one ot the following operations applied to s_1: usv, qqr, rqr, or tra, depending on the chosen inheritance.

(2) Inheritance from two relations. There are also four different cases of origins inheritance: (c_1,c_2), (r_1,r_2), (r_1,c_2), and (c_1,r_2). The other possible combinations, e.g., (c_2,r_1), can be reduced to the aforementioned by swapping s_1 and s_2. Expression tra$\circ$mmu and operations cpd, mmu, and opd applied to s_1 and s_2 cover all four combinations, depending on the chosen inheritance. □

Example 35. Consider relation $v1$ from Figure 4.6. Relation $v1$ is randomly initialized with the SQL query in Figure 4.14.

```
1  SELECT V, uniform [0, 1] AS Y
2  FROM CPD[V]( ( SELECT P, A, B, C, G, T
3                 FROM p ) AS p1 BY P,
4               ( SELECT P, Y
5                 FROM p ) AS p2 BY P
6             );
```

Figure 4.14: Random initialization with cpd

The number of tuples in $v1$ is equal to the number of attributes in $p1$, and the number of attributes in $v1$ is equal to the number of attributes in $p2$. Thus, we use the cpd operation with shape type (c_1,c_2) to create the contextual information forthe randomly initialized relation $v1$. □

Example 36. Consider relation v from Figure 4.16. Relation v is randomly initialized with the SQL expression in Figure 4.15.

```
1  SELECT V, 0 AS A, 1 AS B, 2 AS C, 3 AS G, 4 AS T, 5 AS Y
2  FROM RQR[V]( p BY P );
```

Figure 4.15: Random initialization with rqr

Relation v has origins that are taken from relation p from Figure 4.1. The number of attributes and the number of tuples in v are equal to the number of attributes in application schema in p. Therefore, we create relation v with relational matrix operation rqr that has shape type (c_1,c_1),

□

v

V	A	B	C	G	T	Y
A	0	1	2	3	4	5
B	0	1	2	3	4	5
C	0	1	2	3	4	5
G	0	1	2	3	4	5
T	0	1	2	3	4	5
Y	0	1	2	3	4	5

Figure 4.16: Relation randomly initialized with `rqr`

4.8.2 Shape Preserving Iterations

Lemma 2. *Let $Q^r(r,..) = r'$ be a stable query and r be a randomly initialized relation. Relation r' has a row origin.*

Proof. By Definition 7 of a stable query, Q^r has shape type (r_1, c_1), because r' has the same size as r. Then, according to the definition of origins given in [DAB20b], row origin of r' is $r.\mathbf{U}$ where $\mathbf{U} \subset sch(r), sch(r')$ and is a superkey. By Definition 7, there exists a set of attributes $A \subset sch(r), sch(r')(r.A = r'.A)$ that form a superkey. Thus, $r'.A$ satisfies the definition of origins and is a row origin of r'. □

Theorem 2. *Consider a shape preserving iteration $I_r(Q^r(r,...), E(r,...)) = r'$ and let r be a randomly initialized relation. Result relation r' has row and column origins.*

Proof. Lemma 2 proves that Q^r returns a relation with a row origin. Since each step of the shape preserving iteration is a stable query, which preserves the row origin, the result relation also has the row origin. The column origin of r' is preserved because I_r does not change the schema of r, and the column origin is a part of the schema. □

Example 37. Consider the solution of our application scenario. The relational matrix expression from Figure 4.9, which performs logistic regression, is applied to input relation $v1$ from Figure 4.6 and yields result relation $v1$ shown in Figure 4.2a. Relation $v1$ is a randomly initialized relation and, thus, by Definition 8 includes row and column origins. Result relation $v1$ also has origins, as it is produced by a shape preserving iteration. The origins are essential in the second step of the computation where the risk is estimated. Consider the relational matrix expression in Figure 4.3 that computes relation $v4$. The row origin of relation $v1$, i.e., values of attribute V, is used to establish the proper order of tuples for the relational matrix multiplication. □

4.9 Implementation

In this section we explain our integration of shape preserving iterations into MonetDB. The integration enables iterations over data that is stored in relations and include contextual information.

4.9.1 MonetDB

MonetDB stores and processes all relations as lists of BATs (binary association tables). One BAT represents one attribute in a relation and consists of two one-dimensional arrays: The first array stores OIDs (tuple identifiers) and the second array stores actual attribute values. All values of a tuple have the same OIDs. For example, relation p in Figure 4.1 is stored as a list of seven BATs: One BAT for each attribute, i.e., P, A, C, T, B, G, Y. Each relational operation in MonetDB is represented as a sequence of operations between BATs, such as BAT addition, multiplication and sum computation. When an SQL query is submitted, MonetDB parses, optimizes it, and builds a *statement tree*.

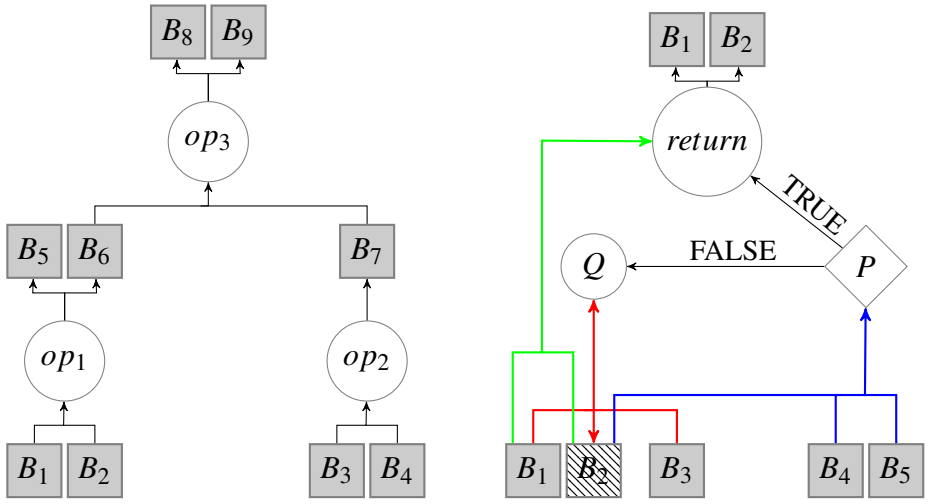
A statement tree is a query tree, where each leaf node refers to a BAT or a constant value, and each inner node is a logical operation over BATs and constants. For example, logical operations include normalization, addition, and rename: $norm(B_1)$, $add(B_1, B_2)$, $rename(B_1, {}'B_2')$.

Figure 4.17a illustrates a standard statement tree. The round nodes are operations over BATs and the gray rectangular nodes are input and output BATs.

The statement tree is interpreted as a *MAL plan*, which is optimized and executed. A MAL plan is a sequence of MonetDB Assemble Language instructions that correspond to physical BAT operations. For a given logical operation there may exist multiple physical operations. For example, logical BAT addition may correspond to physical BAT addition with different arguments, e.g., $mal\_add(B_1[dbl], B_2[dbl])$ or $mal\_add(B_1[int], B_2[int])$[1], depending on the types of the BAT values.

Each statement node is translated to a MAL instruction(s) (i.e., a physical operation) starting from the leftmost bottom operation. After translating the leftmost bottom operation and its siblings, their parent operation is translated to MAL instruction(s). This process is repeated recursively. Algorithm 7 corresponds to the statement tree shown in Figure 4.17a. For example, the

[1] We use prefix *mal_* to denote a physical operation.



(a) Standard **(b)** With iteration

Figure 4.17: Statement trees

leftmost bottom node op_1 is translated to physical operation mal_op_1, then op_2 and op_3 are also translated to the corresponding physical operations.

Algorithm 7: MALPlan(Figure 4.17a)

1 $B_5, B_6 := mal\_op_1(B_1, B_2)$;
2 $B_7 := mal\_op_2(B_3, B_4)$;
3 $B_8, B_9 := mal\_op_3(B_6, B_7)$;

4.9.2 Statement Tree for Shape Preserving Iterations

To integrate shape preserving iterations, we extend a standard statement tree, i.e., used for SELECT queries, with an *update edge* and a *control node*. To implement the control node we leverage MAL plan control instructions that allow to mark a certain place in a plan or jump to a marked place.

Our goal is to provide a statement tree with a shape preserving iteration that can be easily integrated into a bigger tree, i.e., similarly to a standard statement tree it should take input BATs and should output result BATs.

Figure 4.17b illustrates a statement tree with a shape preserving iteration. The tree includes the new update edge (the bidirectional red edge between Q and B_2) and the control node (diamond shaped node P). BAT B_1 is the order part and BAT B_2 is the application part of the iterated

relation, node Q corresponds to iteration body Q, and node P corresponds to exit condition E. The bidirectional edge from node Q to hatched BAT B_2 denotes that B_2 is updated after Q is computed. The diamond shaped control node P passes control to Q or to the *return* node depending on the evaluation of P. The tree returns the BATs of the iterated relation with the updated application part (B_2). The colored edges in Figure 4.17b illustrate the groups of input BATs for different nodes. The green edges connect input BATs B_1 and B_2 with the return node. BATs that are required to compute iteration body Q are connected with node Q by red edges. The input BATs for exit condition P are connected to node P by blue edges.

An extended statement tree has a different flow from a standard statement tree. We consider the flow of execution of the statement tree from Figure 4.17b. First Q is evaluated, updating BAT B_2: Because of the update edge, hatched BAT B_2 is at the same time the output BAT of iteration body Q. BAT B_2 must be updated before other nodes can take it as an input. After that, control node P is evaluated. Control node P does not have output BATs, and it passes control to Q or *return* node, depending on the evaluation of P. The execution of operations in nodes Q and P is repeated until result BATs are returned.

Algorithm 8 corresponds to the statement tree in Figure 4.17b. Iteration body Q is translated to a sequence of physical operations *mal_Q*. *mal_Q* updates BAT B_2 with the new values. Then control node P is translated to if-then-else statement that includes the execution of *mal_P* (similar to *mal_Q*, *mal_P* is a physical implementation of P). The if-then-else statement evaluates the exit condition and passes control to either Q or *return*.

Algorithm 8: MALPlan(Figure 4.17b)

```
1 B2 := mal_Q(B1,B2,B3) ;
2 if mal_P(B2,B4,B5) then
3 |   goto 1;
4 else
5 |   goto 6;
6 return(B1,B2);
```

Example 38. Consider iterated relation r with schema (A,B) and a shape preserving iteration with stable query $\pi_{A,B/2}(r) = r'$ and predicate $\vartheta_{SUM(B)}(r) < 1$. Figure 4.18 illustrates the statement tree with this shape preserving iteration.

First, $div(B,2)$ divides BAT B values by 2, and the initial values in B are updated with the new values. Second, $SUM(B)$ over updated BAT B is computed, yielding BAT S. Third, the control node evaluates $< (S,1)$ and passes the flow back to $div(B,2)$ or to the *return* node, depending on

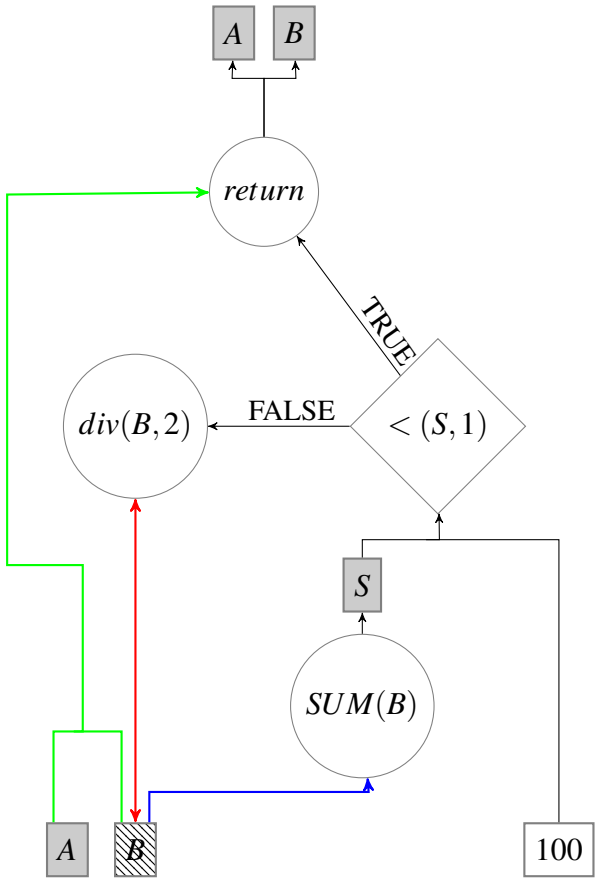


Figure 4.18: Example of a statement tree with a shape preserving iteration

the result. The colored edges denote the same groups of BATs as in Figure 4.17b. This statement tree yields result BATs A and B. They can be used in further calculations as input BATs. □

4.10 Optimization

The integration of shape preserving iterations into a statement tree opens up possibilities for different types of optimizations. We consider some of those optimizations with examples below.

Existing Relational Optimization Both an iteration body and an exit condition in shape preserving iterations are expressed as standard statement trees. Thus, our integration allows existing relational optimizations within iteration bodies and exit conditions of shape preserving iterations.

Consider shape preserving iteration $I_r(Q^r, E)$. Then for any relational optimization rewriting rule O, such as join reordering, the following equivalence holds: $I_r(Q^r, E) \equiv I_r(O(Q^r), O(E))$.

Optimization in Stable Queries A stable query used in an iteration body of a shape preserving iteration always returns a result relation with a known number of tuples. The process of choosing an algorithm to perform the outer (last) operation of the stable query does not require to use heuristics to estimate lower and upper bounds of the result size.

Consider iteration body $Q^r = r'$, whose outer operation is a selection. Since Q^r is a stable query, the number of tuples in r' is equal to the number of tuples in r. This property allows to determine the best strategy to perform the selection, for example, when there is an index on the selected attributes.

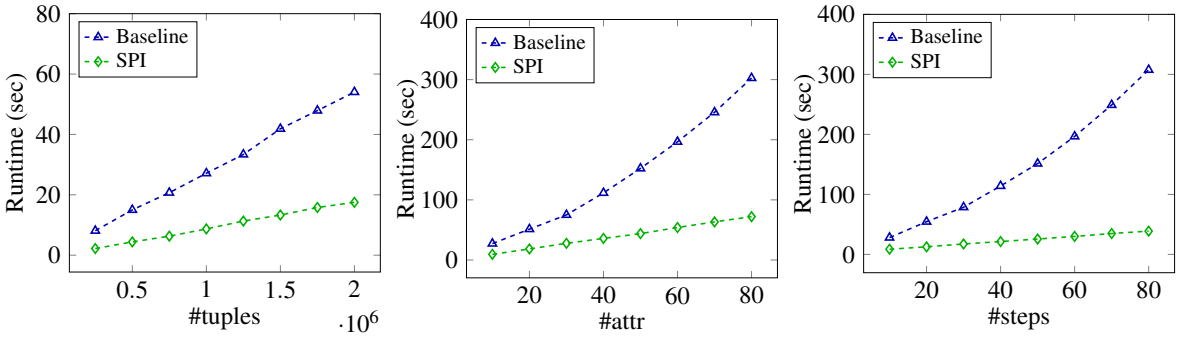
4.11 Evaluation

Setup We integrate shape preserving iterations in MonetDB v11.23.13. We run the evaluation on a virtual machine in the UZH ScienceCloud [Uni20] with Ubuntu 14.04.5 LTS, Intel Haswell processor 2.593GHz (L1: 32K+32K, L2: 4096K) with 4 VCPU, and 15.6GB of RAM. Both server and client are running on the same machine.

Baseline integration of iterations The baseline integration flattens iterations in the statement tree. Exit conditions are not considered in this approach because predicates cannot be checked dynamically without control structures. Thus, the number of iteration steps must be predefined. The statement tree is composed of statement subtrees, which represent the iteration body of a shape preserving iteration, and are repeated the fixed number of times. The baseline approach does not scale because a query statement tree grows very fast due to many repeated iteration body parts.

Comparison We compare our implementation of shape preserving iterations, denoted as SPI, with the baseline integration of iterations in MonetDB. We use synthetic data in our evaluation: Relations include numerical attributes and are populated with random integers in the range from 0 to 150'000.

Figure 4.19 illustrates the runtimes of SPI and the baseline approach for a shape preserving iteration with a varying number of steps and over relations of different sizes. The iteration body multiplies values in all but the first attribute of the iterated relation by 0.9. The exit condition checks if the maximum value in the second attribute is greater than 1.



(a) Varying #tuples **(b)** Varying #attributes **(c)** Varying #steps

Figure 4.19: Runtimes of SPI and the baseline approach for varying number of tuples, attributes, and iterations

To make a fair comparison, we fix the number of performed iteration steps in both SPI and the baseline approach. Thus, the SPI approach always performs the same number of steps as the baseline approach.

The runtime of SPI includes the executions of the iteration body and the exit condition in each step, while the runtime of the baseline approach includes only the execution of the iteration body. Figure 4.19a illustrates the runtimes for a varying number of tuples with 10 iterations and 10 attributes. Figure 4.19b includes the runtimes for iterations a varying number of attributes with 10 iterations and 1M tuples. Finally, Figure 4.19c illustrates the runtimes for iterations with 1M tuples and 10 attributes.

In all three cases SPI outperforms the baseline approach. There are two reasons for that. First, a statement tree in the baseline approach grows with the number of steps. For example, a statement tree of the query from Figure 4.19c with 80 steps includes at least 2400 nodes. As the statement tree grows more resources are needed to create, manage, and interpret it as a MAL plan. Thus, the baseline approach does not scale with the number of iteration steps. Second, the execution of each operation in the baseline approach allocates memory for a resulting BAT, while in the SPI approach memory is allocated once in the first step of the iteration, and then the existing BATs of the intermediate results are updated with the new values.

CHAPTER 5

Conclusion and Future Work

In this thesis we address the problem of combining the linear and the relational algebras into one model that operates on relations with contextual information. We offer a principled solution that removes the mismatch between relations and matrices on the logical level. Thus, our idea of preserving contextual information throughout analytical processing is neither system nor implementation specific and can be adopted by many existing databases. We offer and evaluate an implementation in MonetDB to confirm the feasibility and effectiveness of the proposed approach.

To close the gap between relations and matrices we introduce the concept of contextual information. We identify parts of a relation that are equally important, but have different semantics and, as a consequence, should be processed differently. We offer the relational matrix algebra that is defined over relations. The relational matrix algebra allows to apply linear algebra operations to relations in a straightforward manner and to combine operations from the linear and the relational algebras in one expression. Through the new concept of origins we offer a systematic approach to identify and preserve sufficient contextual information in a result relation. Thus, the result relation: (1) is interpretable, (2) is connected to input relations, and (3) can be used in further calculations.

We define shape preserving iterations over relations, extending the set of available data analysis techniques with iterative methods. Shape preserving iterations are in-place iterations that refine values in an iterated relations and allow to perform tasks such as gradient descent over data stored in relations. We integrate shape preserving iterations into the relational matrix algebra and SQL. We define random initialization: An approach that creates input relations with contextual information for shape preserving iterations. We prove that shape preserving iterations over randomly initialized relations yield result relations with origins.

Finally, we offer an integration of our approach and solution into the column-oriented database MonetDB. The integration relies on leveraging internal MonetDB data structures and algorithms. We evaluate our implementation for various use-cases. We compare it with the state-of-the-art solutions and show that our integration outperforms existing approaches for mixed query workloads, i.e., for queries that combine operations from both algebras. In summary, we offer an extended database system that efficiently supports the combination of relational and analytical tasks.

Future Work In our work we leverage the declarative paradigm to support relational optimizations and deliver an efficient solution for analytical processing over relations. However, data scientists often have no experience with declarative languages and prefer to explore data interactively. They are ready to trade system optimizations for immediate access to the result of each single computation step. One of the possible directions for future work includes developing a solution that supports easy access to intermediate results without losing relational optimizations.

The introduced relational matrix algebra opens up opportunities for exploring and identifying new inter-algebra optimization rules. Optimizations available for shape preserving iterations can also be developed further. Additionally, new equivalence rules that leverage properties of stable queries can be introduced.

Yet another direction for future work is driven by the inability of existing database systems to handle relations with many columns (e.g., hundreds of thousands). This issue can be addressed in different ways. For example, one can investigate approaches that prevent creation of wide intermediate result relations. Another option is to adapt the existing data structures and algorithms to handle wide relations.

Bibliography

[ALOR18] C. R. Aberger, A. Lamb, K. Olukotun, and C. Re. LevelHeaded: A unified engine for business intelligence and linear algebra querying. In *ICDE. IEEE*, 2018.

[ATOR16] C. R. Aberger, S. Tu, K. Olukotun, and C. Re. EmptyHeaded: A relational engine for graph processing. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 431–446. ACM, 2016.

[Aus11] P. Austin. Optimal caliper widths for propensity-score matching when estimating differences in means and differences in proportions in observational studies. *Pharmaceutical statistics*, 10:150–61, 2011.

[BDF+98] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The Multidimensional Database System RasDaMan. *SIGMOD Rec.*, 27(2), 1998.

[BKYJ19] M. Boehm, A. Kumar, J. Yang, and H. V. Jagadish. *Data Management in Machine Learning Systems*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2019.

[Bla06] J. Blackledge. Digital signal processing: Mathematical and computational methods, software development, and applications. 03 2006.

[BLB+13] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly,

B. Holt, and G. Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[BRFR12] C. Binnig, R. Rehrmann, F. Faerber, and R. Riewe. FunSQL: It is time to make SQL functional. In *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, EDBT-ICDT'12, pages 41–46. ACM, 2012.

[Byr08] C.L. Byrne. *Applied iterative methods*. Ak Peters Series. AK Peters, 2008.

[CVP+13] Z. Cai, Z. Vagena, L. Perez, S. Arumugam, P. J. Haas, and C. Jermaine. Simulation of database-valued Markov chains using SimSQL. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 637–648. ACM, 2013.

[DAB20a] O. Dolmatova, N. Augsten, and M. H. Böhlen. Preserving contextual information in relational matrix operations. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pages 1894–1897. IEEE, 2020.

[DAB20b] O. Dolmatova, N. Augsten, and M. H. Böhlen. A relational matrix algebra and its implementation in a column store. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2573–2587. ACM, 2020.

[DDK19] J. V. D'silva, F. De Moor, and B. Kemme. Keep your host language object and also query it: A case for SQL query support in RDBMS for host language objects. In Carlos Maltzahn and Tanu Malik, editors, *Proceedings of the 31st International Conference on Scientific and Statistical Database Management, SSDBM 2019, Santa Cruz, CA, USA, July 23-25, 2019*, pages 133–144. ACM, 2019.

[DDMK18] J. V. D'silva, F. De Moor, and B. Kemme. AIDA: Abstraction for advanced in-database analytics. *Proc. VLDB Endow.*, 11(11):1400–1413, 2018.

[Gan80] W. Gander. Algorithms for the QR-Decomposition. Technical report, ETH Zurich, 1980.

[GKP+11] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. SystemML: Declarative machine learning on MapReduce. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 231–242. IEEE Computer Society, 2011.

[GVL96] G. H. Golub and C. F. Van Loan. *Matrix Computations*. John Hopkins University Press, 3rd edition, 1996.

[HHS17] D. Hutchison, B. Howe, and D. Suciu. LaraDB: A minimalist kernel for linear and relational algebra computation. *CoRR*, abs/1703.07342, 2017.

[HJ19] F. E. Harrell Jr. Right Heart Catheterization Dataset. http://biostat.mc.vanderbilt.edu/wiki/pub/Main/DataSets/rhc.csv, 2019.

[HRS+12] J. M. Hellerstein, C. Re, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib Analytics Library: Or MAD Skills, the SQL. *Proc. VLDB Endow.*, 5(12):1700–1711, 2012.

[Inc19] Kaggle Inc. BIXI Montreal (public bicycle sharing system). https://www.kaggle.com/aubertsigouin/biximtl, 2019.

[Int20] Intel. MKL – Intel Math Kernel Library. https://software.intel.com/en-us/mkl, 2020.

[JLY+19] D. Jankov, S. Luo, B. Yuan, Z. Cai, J. Zou, C. Jermaine, and Z. J. Gao. Declarative recursive computation on an RDBMs: Or, why you should use a database for distributed machine learning. *Proc. VLDB Endow.*, 12(7):822–835, 2019.

[JW07] R. A. Johnson and D. W. Wichern. *Applied Multivariate Statistical Analysis*. Applied Multivariate Statistical Analysis. Pearson Prentice Hall, 2007.

[LGG+17] S. Luo, Z. J. Gao, M. Gubanov, L. L. Perez, and C. Jermaine. Scalable linear algebra on a relational database system. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 523–534, 2017.

[MB15] D. Misev and P. Baumann. Homogenizing data and metadata retrieval in scientific applications. In *Proceedings of the ACM Eighteenth International Workshop on Data Warehousing and OLAP*, DOLAP '15, pages 25–34. ACM, 2015.

[McK11] W. McKinney. pandas: a foundational python library for data analysis and statistics, 2011.

[MJ09] J. H. Martin and D. Jurafsky. *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition*. Pearson/Prentice Hall Upper Saddle River, 2009.

[MM99] O. L. Mangasarian and D. R. Musicant. Data discrimination via nonlinear generalized support vector machines. Technical report, Complementarity: Applications, Algorithms and Extensions, 1999.

[Mon17] MonetDB. Online MonetDB reference. https://www.monetdb.org/Home, 2017.

[Ng04] A. Y. Ng. Feature selection, l1 vs. l2 regularization, and rotational invariance. In *Proceedings of the twenty-first international conference on Machine learning*, page 78. ACM, 2004.

[Num18] NumPy. http://www.numpy.org, 2018.

[NVI19] NVIDIA. cuBLAS – NVIDIA BLAS library for GPUs. https://developer.nvidia.com/cublas, 2019.

[Ora16] Oracle. Online Oracle UTL_NLA Package Reference. https://docs.oracle.com/database/121/ARPLS/u_nla.htm, 2016.

[Ord07] C. Ordonez. Building statistical models and scoring with UDFs. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 1005–1016. ACM, 2007.

[Pro18] R Project. The R Project for Statistical Computing. https://www.r-project.org/, 2018.

[Pro19] R Project. The R Stats Package: R statistical functions. https://www.rdocumentation.org/packages/stats, 2019.

[PTH+17] L. Passing, M. Then, N. Hubig, H. Lang, M. Schreier, S. Günnemann, A. Kemper, and T. Neumann. SQL- and operator-centric data analytics in relational main-memory databases. In *Proceedings of the 20th International Conference on Ex-*

tending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017, pages 84–95, 2017.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[QR17] Quick R. R Matrix Algbera package overview. http://www.statmethods.net/advstats/matrix.html, 2017.

[RR83] P. R. Rosenbaum and D. B. Rubin. The central role of the propensity score in observational studies for causal effects. *Biometrika*, 70(1):41–55, 1983.

[RRT95] C. Radhakrishna Rao and H. Toutenburg. *Linear Models, Least Squares and Alternatives*. Springer Series in Statistics. Springer-Verlag New York, 1995.

[Rud16] S. Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[SBPR11] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman. The Architecture of SciDB. In *Proceedings of the 23rd International Conference on Scientific and Statistical Database Management*, SSDBM'11, pages 1–16. Springer-Verlag, 2011.

[Sci13] Inc. SciDB. SciDB User's Guide Version 13.3.6203. In *SciDB User's Guide*, pages 1–258, 2013.

[SGC07] S. Senn, E. Graf, and A. Caputo. Stratification for the propensity score compared with linear regression techniques to assess the effect of treatment or exposure. *Statistics in Medicine*, 26(30):5529–5544, 2007.

[Uni20] University of Zurich. ScienceCloud. https://www.zi.uzh.ch/en/teaching-and-research/science-it/infrastructure/, 2020.

[UoT19] University of Trier. DBLP computer science bibliography. https://dblp.uni-trier.de, 2019.

[Var62] R. S. Varga. *Matrix Iterative Analysis*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, 1962.

[YS09] X. Yan and X. G. Su. *Linear Regression Analysis: Theory and Computing*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2009.

[ZHY09] Y. Zhang, H. Herodotou, and J. Yang. RIOT: I/O-efficient numerical computing without SQL. *CoRR*, abs/0909.1766, 2009.

[ZKIN11] Y. Zhang, M. Kersten, M. Ivanova, and N. Nes. SciQL: Bridging the Gap Between Science and Relational DBMS. In *Proceedings of the 15th Symposium on International Database Engineering & Applications*, IDEAS '11. ACM, 2011.

[ZKM13] Y. Zhang, M. Kersten, and S. Manegold. SciQL: Array Data Processing Inside an RDBMS. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1049–1052. ACM, 2013.

OKSANA DOLMATOVA

PERSONAL DETAILS

Date of Birth: 10 Feb 1991
Nationality: Russian

EDUCATION

Jan 2014 - Feb 2021 **Doctoral Program** at the Universität Zürich, Department of Informatics
Database Technology Group, Prof. Dr. Michael Böhlen

Sep 2008 - Jun 2013 **Mathematics and Mechanics Faculty, Saint Petersburg State University**
Saint Petersburg, Russia
5-year Diploma (joint BSc and MSc)
Thesis: *"An Adaptive Approximate Algorithm For Join"*
Thesis Grade: *5/5*, Supervisor: Prof. Boris Novikov

PROFESSIONAL EXPERIENCE

Jan 2014 - Jul 2020 **Department of Informatics, Universität Zürich** Zürich, Switzerland
Research Assistant, Database Technology Group

Feb 2012 - Dec 2013 **Mathematics and Mechanics Faculty, Saint Petersburg State University**
Saint Petersburg, Russia
Research Assistant, Operations Research Laboratory

Jul 2011 - Dec 2013 **Luxoft** Saint Petersburg, Russia
Junior developer, Technology Strategy Center